TECHNICAL REPORT RD-GC-93-36

REAL-TIME EXECUTIVE FOR MILITARY SYSTEMS C APPLICATIONS USER'S GUIDE

Wanda M. Hughes
Guidance and Control Directorate
Research, Development, and Engineering Center

and

On-Line Applications Research
2227 Drake Avenue, Suite 10-F
Huntsville, AL 35805

November 1993

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35898-5000

Approved for Public Release; Distribution is Unlimited.

DESTRUCTION NOTICE

FOR CLASSIFIED DOCUMENTS, FOLLOW THE PROCEDURES IN DoD 5200.22-M, INDUSTRIAL SECURITY MANUAL, SECTION II-19 OR DoD 5200.1-R, INFORMATION SECURITY PROGRAM REGULATION, CHAPTER IX. FOR UNCLASSIFIED, LIMITED DOCUMENTS, DESTROY BY ANY METHOD THAT WILL PREVENT DISCLOSURE OF CONTENTS OR RECONSTRUCTION OF THE DOCUMENT.

DISCLAIMER

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

TRADE NAMES

USE OF TRADE NAMES OR MANUFACTURERS IN THIS REPORT DOES NOT CONSTITUTE AN OFFICIAL ENDORSEMENT OR APPROVAL OF THE USE OF SUCH COMMERCIAL HARDWARE OR SOFTWARE.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188 Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-RD-GC-93-36	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Guidance and Control Directorate RD&E Center 6b. OFFICE SYMBOL (If applicable) AMSMI-RD-GC-S	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Commander, U.S. Army Missile Command ATTN: AMSMI-RD-GC-S Bldg. 4381 (Wanda Hughes) Redstone Arsenal, AL 35898-5254	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Real-Time Executive for Military Systems C Applications User's Guide

12. PERSONAL AUTHOR(S)
Wanda M. Hughes

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM June 89 TO Jan 93	November 1993	196

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	RTEMS	Real-Time	executive
			C Language	Intertask Communication	heterogeneous
			Synchronization	Semaphore	scheduling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document is a user's manual for a real-time multiprocessor executive which provides a high performance environment for embedded military applications including such features as multitasking capabilities; homogeneous and heterogeneous multiprocessor systems; time event-driven, priority-based, preemptive scheduling; intertask communication and synchronization; responsive interrupt management; dynamic memory allocation; and a high level of user configurability. This executive, known as RTEMS (Real-Time Executive for Missile Systems) was originally developed in an effort to eliminate many of the major drawbacks of the Ada programming language. RTEMS is based on the RTEMS (now ORKID) proposed standard. The code is Government owned, so no licensing fees are necessary. The executive is written using the 'C' programming language with a very small amount of assembly language code. The code was developed as a linkable and/or ROMable library with the Ada programming language. Initially RTEMS was developed for the Motorola 68000 family of processors. It has been ported to the Intel 80386 and 80960 families. Other processor ports are planned for the future. RTEMS documents and code are available free of charge by contacting RTEMS, U. S. Army Missile Command, ATTN: AMSMI-RD-GC-S, Redstone Arsenal, AL 35898-5254.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Wanda M. Hughes	22b. TELEPHONE (Include Area Code) (205) 876-4484	22c. OFFICE SYMBOL AMSMI-RD-GC-S

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PREFACE

In recent years, the cost required to develop a software product has increased significantly while the target hardware costs have decreased. Now a larger portion of money is expended in developing, using, and maintaining software. The trend in computing costs is the complete dominance of software over hardware costs. Because of this, it is necessary that formal disciplines be established to increase the probability that software is characterized by a high degree of correctness, maintainability, and portability. In addition, these disciplines must promote practices that aid in the consistent and orderly development of a software system within schedule and budgetary constraints. To be effective, these disciplines must adopt standards which channel individual software efforts toward a common goal.

The push for standards in the software development field has been met with various degrees of success. The Microprocessor Operating Systems Interfaces (MOSI) effort has experienced only limited success. As popular as the **UNIX** operating system has grown, the attempt to develop a standard interface definition to allow portable application development has only recently begun to produce the results needed in this area. Unfortunately, very little effort has been expended to provide standards addressing the needs of the real–time community. Several organizations have addressed this need during the past several years.

The Real–Time Executive Interface Definition (RTEID) was developed by Motorola with technical input from Software Components Group. **RTEID** was adopted by the VMEbus International Trade Association (**VITA**) as a baseline draft for their proposed standard multiprocessor, real–time executive interface, Open Real–Time Kernel Interface Definition (**ORKID**). These two groups are currently working together with the IEEE P1003.4 committee to insure that the functionality of their proposed standards is adopted as the real–time extensions to **POSIX**.

This emerging standard defines an interface for the development of real–time software to ease the writing of real–time application programs that are directly portable across multiple real–time executive implementations. This interface includes both the source code interfaces and run–time behavior as seen by a real–time application. It does not include the details of how a kernel implements these functions. The standard's goal is to serve as a complete definition of external interfaces so that application code that conforms to these interfaces will execute properly in all real–time executive environments. With the use of a standards compliant executive, routines that acquire memory blocks, create and manage message queues, establish and use semaphores, and send and receive signals need not be redeveloped for a different real–time environment as long as the new environment is compliant with the standard. Software developers need only concentrate on the hardware dependencies of the real–time system. Furthermore, most hardware dependencies for real–time applications can be localized to the device drivers.

A compliant executive provides simple and flexible real–time multiprocessing. It easily lends itself to both tightly–coupled and loosely–coupled configurations (depending on the system hardware configuration). Objects such as tasks, queues, events, signals, semaphores, and memory blocks can be designated as global objects and accessed by any task regardless of which processor the object and the accessing task reside.

The acceptance of a standard for real–time executives will produce the same advantages enjoyed from the push for **UNIX** standardization by AT&T's System V Interface Definition and **IEEE's POSIX** efforts. A compliant multiprocessing executive will allow close coupling

between UNIX systems and real–time executives to provide the many benefits of the UNIX development environment to be applied to real–time software development. Together they provide the necessary laboratory environment to implement real–time, distributed, embedded systems using a wide variety of computer architectures.

A study was completed in 1988, within the Research, Development, and Engineering Center, U.S. Army Missile Command, which compared the various aspects of the Ada programming language as they related to the application of Ada code in distributed and/or multiple processing systems. Several critical conclusions were derived from the study. These conclusions have a major impact on the way the Army develops application software for embedded applications. These impacts apply to both in–house software development and contractor developed software.

A conclusion of the analysis, which has been previously recognized by other agencies attempting to utilize Ada in a distributed or multiprocessing environment, is that the Ada programming language does not adequately support multiprocessing. Ada does provide a mechanism for multi–tasking; however, this capability exists only for a single processor system. The language also does not have inherent capabilities to access global named variables, flags, or program code. These critical features are essential in order for data to be shared between processors. However, these drawbacks do have *workarounds* which are sometimes awkward and defeat the intent of software maintainability goals.

Another conclusion drawn from the analysis, was that the run–time executives being delivered with the Ada compilers were too slow and inefficient to be used in modern missile systems. A run–time executive is the core part of the run–time system code, or operating system code, that controls task scheduling, input/output management and memory management. Traditionally, whenever efficient executive (also known as kernel) code was required by the application, the user developed in–house software. This software was usually written in assembly language for optimization.

Because of this shortcoming in the Ada programming language, software developers in research and development and contractors for project managed systems, are mandated by technology to purchase and utilize off–the–shelf third party kernel code. The contractor, and eventually the Government, must pay a licensing fee for every copy of the kernel code used in an embedded system.

The main drawback to this development environment is that the Government does not own, nor has the right to modify code contained within the kernel. V&V techniques in this situation are more difficult than if the complete source code were available. Responsibility for system failures due to faulty software is yet another area to be resolved under this environment.

The Guidance and Control Directorate began a software development effort to address these problems. A project to develop an experimental run–time kernel was begun that will eliminate the major drawbacks of the Ada programming language mentioned above. The Real–Time Executive for Military Systems (RTEMS) provides full capabilities for management of tasks, interrupts, time, and multiple processors in addition to those features typical of generic operating systems. The code is Government owned, so no licensing fees are necessary. The code was developed as a linkable and/or ROMable library with the Ada programming language. Initially the library code was developed on the Motorola 68000 family of processors using the C programming language as the development language. It has since been ported to the Intel i80386 and Intel i80960 families. Other language interfaces and processor families, including RISC, CISC, and DSP, are planned in the future.

The RTEMS multiprocessor support is capable of handling either homogeneous or heterogeneous systems. The kernel automatically compensates for architectural differences (byte swapping, etc.) between processors. This allows a much easier transition from one processor family to another without a major system redesign.

Since the proposed standards are still in draft form, RTEMS cannot and does no claim compliance. However, the status of the standard is being carefully monitored to guarantee that RTEMS provides the functionality specified in the standard. Once approved, RTEMS will be made compliant.

This document is detailed design guide for a functionally compliant real–time multiprocessor executive. It describes the user interface and run–time behavior of RTEMS.

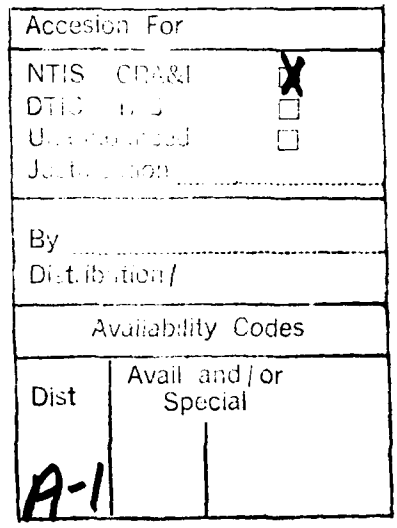

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLLUSTRATIONS

I. OVERVIEW

A. Introduction/Overview

Real–Time Executive for Military Systems, (**RTEMS**) is a real–time executive (kernel) which provides a high performance environment for embedded military applications including the following features:

- *multitasking capabilities*
- *homogeneous and heterogeneous multiprocessor systems*
- *event–driven, priority–based, preemptive scheduling*
- *intertask communications and synchronization*
- *responsive interrupt management*
- *dynamic memory allocation*
- *high level of user configurability*

This manual describes the implementation of **RTEMS** for applications using the C programming language. Those implementation details that are processor dependent are provided in the **C Applications Supplement** documents. A supplement document which addresses specific architectural issues that affect **RTEMS** is provided for each processor type that supported.

The **Assembly Language Interface** definition describes how assembly language routines may utilize **RTEMS** services. This definition is provided in the **Assembly Language Interface** chapter of the **C Applications Supplement** document for the desired target CPU platform.

B. Real–time Application Systems

Real–time application systems are a special class of computer applications. They have a complex set of characteristics that distinguish them from other software problems. Generally, they must adhere to more rigorous requirements. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real–time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints referred to as **deadlines**. Systems can be buried by an avalanche of interdependent, asynchronous or cyclical event streams.

Deadlines can be further characterized as either hard or soft based upon the value of the results when produced after the deadline has passed. A deadline is hard if the results have no value or if their use will result in a catastrophic event. In contrast, results which are produced after a soft deadline may have some value.

Another distinguishing requirement of real–time application systems is the ability to coordinate or manage a large number of concurrent activities. Since software is a synchronous entity, this presents special problems. One instruction follows another in a repeating synchronous cycle. Even though mechanisms have been developed to allow for the processing of external asynchronous events, the software design efforts required to process and manage these events and tasks are growing more complicated.

The design process is complicated further by spreading this activity over a set of processors instead of a single processor. The challenges associated with designing and building real–time application systems become very complex when multiple processors are involved. New requirements such as interprocessor communication channels and global resources that must be shared between competing processors are introduced. The ramifications of multiple processors complicate each and every characteristic of a real–time system.

C. Real–time Executive

Fortunately, real–time operating systems or real–time executives serve as a cornerstone on which to build the application system. A real–time multitasking executive allows an application to be cast into a set of logical, autonomous processes or tasks which become quite manageable. Each task is internally synchronous, but different tasks execute independently, resulting in an asynchronous processing stream. Tasks can be dynamically paused for many reasons resulting in a different task being allowed to execute for a period of time. The executive also provides an interface to other system components such as interrupt handlers and device drivers. System components may request the executive to allocate and coordinate resources, and to wait for and trigger synchronizing conditions. The executive system calls effectively extend the CPU instruction set to support efficient multitasking. By causing tasks to travel through well–defined state transitions, system calls permit an application to demand–switch between tasks in response to real–time events.

By proper grouping of responses to stimuli into separate tasks, a system can now asynchronously switch between independent streams of execution, directly responding to external stimuli as they occur. This allows the system design to meet critical performance specifications which are typically measured by guaranteed response time and transaction throughput. The multiprocessor extensions of **RTEMS** provide the features necessary to manage the extra requirements introduced by a system distributed across several processors. It removes the physical barriers of processor boundaries from the world of the system designer enabling more critical aspects of the system to receive the required attention. Such a system, based on an efficient real–time, multiprocessor executive, is a more realistic model of the outside world or environment for which it is designed. As a result, the system will always be more logical, efficient, and reliable.

By using the directives provided by **RTEMS**, the real–time applications developer is freed from the problem of controlling and synchronizing multiple tasks and processors. In addition, one need not develop, test, debug, and document routines to manage memory, pass messages, or provide mutual exclusion. The developer is then able to concentrate solely on the application. By using standard software components, the time and cost required to develop sophisticated real–time applications is significantly reduced.

D. RTEMS Application Architecture

One important design goal of **RTEMS** was to provide a bridge between two critical layers of typical real–time systems. As shown in Figure 1, **RTEMS** serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real–time applications can be localized to the low level device drivers. The **RTEMS** I/O interface manager provides an efficient tool for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code that accesses them. A well designed real–time system can benefit from this architecture by building a rich library of standard application components which can be used repeatedly in other real–time projects.

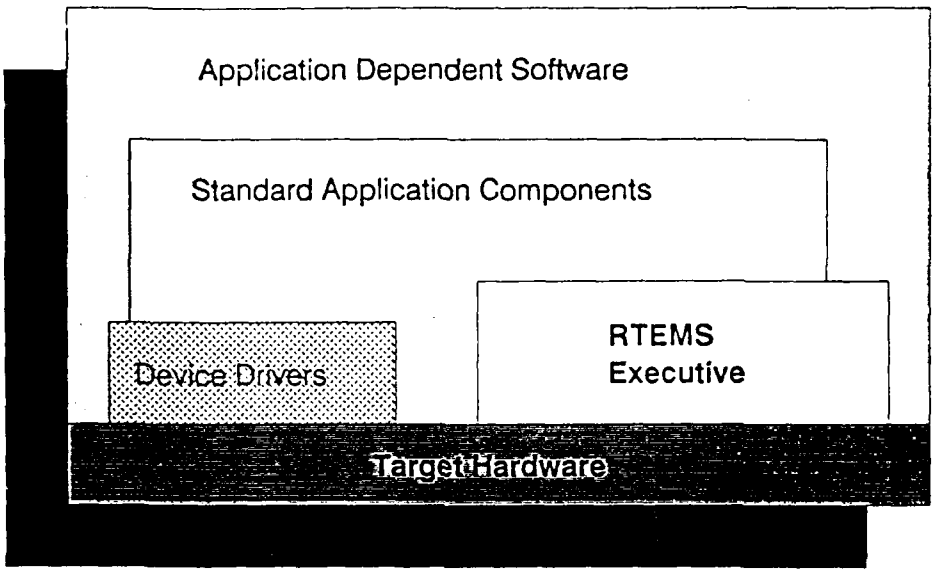

Figure 1. **RTEMS** *Application Architecture*

E. RTEMS Internal Architecture

As illustrated in Figure 2, **RTEMS** can be viewed as a set of components that work in harmony to provide a set of services to a real–time application system. The executive interface presented to the application is formed by grouping directives into logical sets called **resource managers.** *Functions* utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core. Together these components provide a powerful run–time environment that promotes the development of efficient real–time application systems. Subsequent chapters present a detailed description of the capabilities provided by each of the following **RTEMS** managers:

- *initialization*
- *task*
- *interrupt*
- *time*
- *semaphore*
- *message*
- *event*
- *signal*
- *partition*
- *region*
- *dual ported memory*
- *I/O*
- *rate monotonic*
- *fatal error*

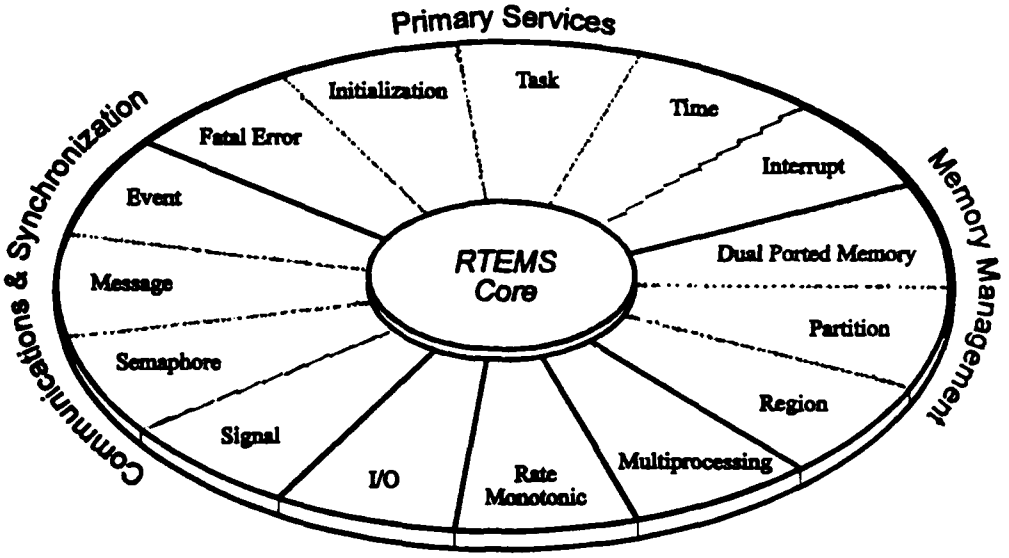

Figure 2. **RTEMS** *Internal Architecture*

The **C Interface Library** component is a collection of routines which allow **C** application programs to invoke **RTEMS** directives. This extends the standard **C** language to include the real–time features provided by **RTEMS** without modifying the **C** compiler.

F. User Customization and Extensibility

As 32–bit microprocessors have decreased in cost, they have become increasingly common in a variety of embedded systems. A wide range of custom and general–purpose processor boards are based on various 32–bit processors. **RTEMS** was designed to make no assumptions concerning the characteristics of individual microprocessor families or of specific support hardware. In addition, **RTEMS** allows the system developer a high degree–of–freedom in customizing and extending its features.

RTEMS assumes the existence of a supported microprocessor and sufficient memory for both **RTEMS** and the real–time application. Board dependent components such as clocks, interrupt controllers, or I/O devices can be easily integrated with **RTEMS**. The customization and extensibility features allow **RTEMS** to efficiently support as many environments as possible.

G. Portability

The issue of portability was the major factor in the creation of **RTEMS**. Since **RTEMS** is designed to isolate the hardware dependencies in the specific board support packages, the real–time application should be easily ported to any other processor. The use of **RTEMS** allows the development of real–time applications which can be completely independent of a particular microprocessor architecture.

H. Memory Requirements

Since memory is a critical resource in many real–time embedded systems, **RTEMS** was specifically designed to allow unused managers to be excluded from the run–time

environment. This allows the application designer the flexibility to tailor **RTEMS** to most efficiently meet system requirements while still satisfying even the most stringent memory constraints. As result, the size of the **RTEMS** executive is application dependent. The **Memory Requirements** chapter of the **C Applications Supplement** document for a specific target processor provides a worksheet for calculating the memory requirements of a custom **RTEMS** run–time environment. The following managers may be optionally excluded:

- *signal*
- *region*
- *dual ported memory*
- *I/O*
- *event*
- *multiprocessing*
- *partition*
- *time*
- *semaphore*
- *message*
- *rate monotonic*

RTEMS utilizes memory for both code and data space. Although **RTEMS'** data space must be in RAM, its code space can be located in either ROM or RAM.

I. Audience

This manual was written for experienced real–time software developers. Although some background is provided, it is assumed that the reader is familiar with the concepts of task management as well as intertask communication and synchronization. Since directives, user related structures, and examples are presented in **C**, and basic understanding of the C programming language is required. A working knowledge of the target processor is helpful in understanding some of **RTEMS'** features. A thorough understanding of the executive cannot be obtained without studying the entire manual because many of **RTEMS'** concepts and features are interrelated. Experienced **RTEMS** users will find that the manual organization facilitates its use as a reference document.

J. Conventions

The following conventions are used in this manual:

- *Significant words or phrases as well as all directive names are printed in* **bold type.**
- *Items in* **bold** *capital letters are constants defined by* **RTEMS.** *Each language interface provided by* **RTEMS** *includes a file containing the standard set of constants, data types, and structure/record definitions which can be incorporated into the user application.*
- *A number of type definitions are provided by* **RTEMS** *and can be found in* **rtems.h.**
- *The characters "***0x***" preceeding a number indicates that the number is in hexadecimal format. Any other numbers are assumed to be in decimal format.*
- *The ampersand character (&) preceeding a symbol indicates that the address of the symbol is passed to the called routine instead of the value itself.*

K. Manual Organization

This first chapter has presented the introductory and background material for the **RTEMS** executive. The remaining chapters of this manual present a detailed description of **RTEMS** and the environment, including run–time behavior, it creates for the user.

A chapter is dedicated to each manager and provides a detailed discussion of each **RTEMS** manager and the directives which it provides. The presentation format for each directive includes the following sections:

- *Calling sequence*
- *Input parameters*
- *Output parameters*
- *Directive status codes*
- *Description*
- *Notes*

The following provides an overview of the remainder of this manual:

Chapter 2: Key Concepts: presents an introduction to the ideas which are common across multiple **RTEMS** managers.

Chapter 3: Initialization Manager: describes the functionality and directives provided by the Initialization Manager.

Chapter 4: Task Manager: describes the functionality and directives provided by the Task Manager.

Chapter 5: Interrupt Manager: describes the functionality and directives provided by the Interrupt Manager.

Chapter 6: Time Manager: describes the functionality and directives provided by the Time Manager.

Chapter 7: Semaphore Manager: describes the functionality and directives provided by the Semaphore Manager.

Chapter8: Message Manager: describes the functionality and directives provided by the Message Manager.

Chapter 9: Event Manager: describes the functionality and directives provided by the Event Manager.

Chapter 10: Signal Manager: describes the functionality and directives provided by the Signal Manager.

Chapter 11: Partition Manager: describes the functionality and directives provided by the Partition Manager.

Chapter 12: Region Manager: describes the functionality and directives provided by the Region Manager.

Chapter 13: Dual–Ported Memory Manager: describes the functionality and directives provided by the Dual–Ported Memory Manager.

Chapter 14: I/O Manager: describes the functionality and directives provided by the I/O Manager.

Chapter 15: Fatal Error Manager: describes the functionality and directives provided by the Fatal Error Manager.

Chapter 16: Scheduling Concepts: details the **RTEMS** scheduling algorithm and task state transitions.

Chapter 17: Rate Monotonic Manager: describes the functionality and directives provided by the Fatal Error Manager.

Chapter 18: Board Support Packages: defines the functionality required of user–supplied board support packages.

Chapter 18: User Extensions: shows the user how to extend **RTEMS** to incorporate custom features.

Chapter 20: Configuring a System: details the process by which one tailors **RTEMS** for a particular single–processor or multiprocessor application.

Chapter 21: Multiprocessing Manager: presents a conceptual overview of the multiprocessing capabilities provided by **RTEMS** as well as describing the **Multiprocessing Communications Interface Layer** and Multiprocessing Manager directives.

Appendix A: Directive Status Codes: provides a definition of each of the directive status codes referenced in this manual. These definitions are also provided in the user include file **rtems.h** which is provided in the **Header Files** chapter of the **C Applications Supplement** for a specific target processor.

Appendix B: Example Application: provides a template for simple **RTEMS** applications.

Appendix C: Glossary: defines terms used throughout this manual.

II. KEY CONCEPTS

A. Introduction

The facilities provided by **RTEMS** are built upon a foundation of very powerful concepts. These concepts must be understood before the application developer can efficiently utilize **RTEMS**. The purpose of this chapter is to familiarize one with these concepts.

B. Objects

RTEMS provides directives which can be used to dynamically create, delete, and manipulate a set of predefined object types. These types include tasks, message queues, semaphores, memory regions, memory partitions, timers, and ports. The object–oriented nature of **RTEMS** encourages the creation of modular applications built upon reusable "building block" routines.

All objects are created on the local node as required by the application and have a **RTEMS** assigned ID. All objects except timers have a user–assigned name. Although a relationship exists between a object's name and its **RTEMS** assigned ID, the name and ID are not identical. Object names are completely arbitrary and selected by the user as a meaningful "tag" which may commonly reflect the object's use in the application. Conversely, object IDs are designed to facilitate efficient object manipulation by the executive.

An **object name** is a unsigned 32–bit entity associated with the object by the user. Although not required by **RTEMS**, object names are typically composed of four ASCII characters which help identify that object. For example, a task which causes a light to blink might be called "LITE". On the other hand, if a application requires 100 tasks, it would be difficult to assign meaningful ASCII names to each task. A more convenient approach would be to name them the binary values 1 through 100, respectively.

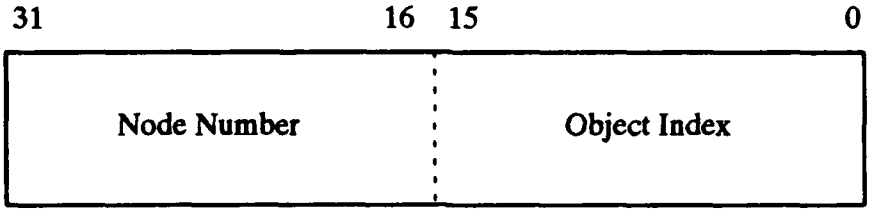

Figure 3. Object ID Composition

An **object ID** is a unique unsigned 32–bit entity composed of two parts. The most significant 16–bits are the number of the node on which this object was created. The node number is always one (1) in a single processor system. The least significant 16–bit form an identifier within a particular object type. This identifier, called the object index, ranges in value from 1 to the maximum number of objects configured for this object type (Fig 3).

The two components of an object ID make it possible to quickly locate any object in even the most complicated multiprocessor system. Object ID's are associated with an object by **RTEMS** when the object is created and the corresponding ID is returned by the appropriate object create directive. The object ID is required as input to all directives involving objects, except those which create an object or obtain the ID of an object.

The object identification directives can be used to dynamically obtain a particular object's ID given its name. This mapping is accomplished by searching the name table associated with this object type. If the name is non–unique, then the ID associated with the first occurrence of the name will be returned to the application. Since object IDs are returned when the object is created, the object identification directives are not necessary in a properly designed single processor application.

An **object control block** is a data structure defined by **RTEMS** which contains the information necessary to manage a particular object type. For efficiency reasons, the format of each object type's control block is different. However, many of the fields are similar in function. The number of each type of control block is application dependent and determined by the values specified in the user's **Configuration Table**. An object control block is allocated at object create time and freed when the object is deleted. With the exception of user extension routines, object control blocks are not directly manipulated by user applications.

C. Communication and Synchronization

In real–time multitasking applications, the ability for cooperating execution threads to communicate and synchronize with each other is imperative. A real–time executive should provide an application with the following capabilities:

- *Data transfer between cooperating tasks*
- *Data transfer between tasks and ISRs*
- *Synchronization of cooperating tasks*
- *Synchronization of tasks and ISRs*

Most **RTEMS** managers can be used to provide some form of communication and/or synchronization. However, managers dedicated specifically to communication and synchronization provide well established mechanisms which directly map to the application's varying needs. This level of flexibility allows the application designer to match the features of a particular manager with the complexity of communication and synchronization required. The following managers were specifically designed for communication and synchronization:

- *Semaphore*
- *Event*
- *Message*
- *Signal*

The semaphore manager supports mutual exclusion involving the synchronization of access to one or more shared user resources. The message manager supports both communication and synchronization, while the event manager primarily provides a high performance synchronization mechanism. The signal manager supports only asynchronous communication and is typically used for exception handling.

D. Time

The development of responsive real–time applications requires an understanding of how **RTEMS** maintains and supports time–related operations. The basic unit of time in **RTEMS** is known as a **tick**. The frequency of clock ticks is completely application dependent and determines the granularity and accuracy of all interval and calendar time operations.

By tracking time in units of ticks, **RTEMS** is capable of supporting interval timing functions such as task delays, timeouts, timeslicing, the delayed posting of events, and the rate monotonic scheduling of tasks. An **interval** is defined as a number of ticks relative to the current time. For example, when a task delays for an interval of ten ticks, it is implied that the task will not execute until ten clock ticks have occurred.

A characteristic of interval timing is that the actual interval period may be a fraction of a tick less than the interval requested. This occurs because the time at which the delay timer is set up occurs at some time between two clock ticks. Therefore, the first countdown tick occurs in less than the compiete time interval for a tick. This can be a problem if the clock granularity is large.

The rate monotonic scheduling algorithm is a hard real–time scheduling methodology. This methodology provides rules which allows one to guarantee that a set of independent periodic tasks will always meet their deadlines – – even under transient overload conditions. The rate monotonic manager provides directives built upon the time manager's interval timer support routines.

Interval timing is not sufficient for the many applications which require that time be kept in wall time or true calendar form. Consequently, **RTEMS** maintains the current date and time. This allows selected time operations to be scheduled at an actual calendar date and time. For example, a task could request to delay until midnight on New Year's Eve before lowering the ball at Times Square.

Obviously, the time manager's directives cannot operate without some external mechanism which provides a periodic clock tick. This clock tick is typically provided by a real time clock or counter/timer device.

E. Memory Management

RTEMS memory management facilities can be grouped into two classes: dynamic memory allocation and address translation. Dynamic memory allocation is required by applications whose memory requirements vary through the application's course of execution. Address translation is needed by applications which share memory with another CPU or an intelligent Input/Output processor. The following **RTEMS** managers provide facilities to manage memory:

- *Region*
- *Dual–Ported Memory*
- *Partition*

RTEMS memory management features allow an application to create simple memory pools of fixed size buffers and/or more complex memory pools of variable size segments. The partition manager provides directives to manage and maintain pools of fixed size entities such as resource control blocks. Alternatively, the region manager provides a more general purpose memory allocation scheme that supports variable size blocks of memory which are dynamically obtained and freed by the application. The dual–ported memory manager provides executive support for address translation between internal and external dual–ported RAM address space.

III. INITIALIZATION MANAGER

A. Introduction

The **initialization manager** is responsible for initiating **RTEMS**, creating and starting all configured initialization tasks, and for invoking the initialization routine for each user–supplied device driver. In a multiprocessor configuration, this manager also initializes the interprocessor communications layer. The directive provided by the initialization manager is:

Name	Directive Description
init_exec	Initialize RTEMS

B. Background

1. Initialization Tasks

Initialization task(s) are the mechanism by which **RTEMS** transfers initial control to the user's application. Initialization tasks differ from other application tasks in that they are defined in the **Initialization Task Table** and automatically created and started by **RTEMS** as part of its initialization sequence. Since the initialization tasks are scheduled using the same algorithm as all other **RTEMS** tasks, they must be configured at a priority and mode which will insure that they will complete execution before other application tasks execute. Although there is no upper limit on the number of initialization tasks, an application is required to define at least one.

A typical initialization task will create and start the static set of application tasks. It may also create any other objects used by the application. Initialization tasks which only perform initialization should delete themselves upon completion to free resources for other tasks. Initialization tasks may transform themselves into a "normal" application task. This transformation typically involves changing priority and execution mode. **RTEMS** does not automatically delete the initialization tasks.

2. The System Initialization Task

The **System Initialization Task** is responsible for initializing all device drivers. As a result, this task has a higher priority than all other application tasks to insure that no application tasks executes until all device drivers are initialized. After device initialization in a single processor system, this task will delete itself.

In multiprocessor configurations, the **System Initialization Task** does not delete itself after initializing the device drivers. Instead it transforms itself into the **Multiprocessing Server** which initializes the **Multiprocessor Communications Interface** layer, verifies multiprocessor system consistency, and processes all requests from remote nodes.

3. The Idle Task

The **Idle Task** is the lowest priority task in all systems and executes only when no other task is ready to execute. This task consists of an infinite loop and will be preempted when any other task is made ready to execute.

4. Initialization Manager Failure

The **k_fatal** directive will be called from **init_exec** for any of the following reasons:

- *If no user initialization tasks are configured. At least one initialization task must be configured to allow* **RTEMS** *to pass control to the application at the end of the executive initialization sequence.*
- *If a* **CPU Dependent Information Table** *is required by the target processor and* **NULL_CPU_TABLE** *is passed to* **init_exec.**
- *If the starting address of the* **RTEMS RAMS Workspace,** *supplied by the application in the* **Configuration Table,** *is not aligned on a four–byte boundary.*
- *If the size of the* **RTEMS RAM Workspace** *is not large enough to initialize and configure the system.*
- *If multiprocessing is configured and the* **node** *entry in the* **Multiprocessor Configuration Table** *is not between one and the* **max_nodes** *entry.*
- *If any of the user initialization tasks cannot be created or started successfully.*

C. Operations

Initializing **RTEMS**

The **init_exec** directive is called by the board support package at the completion initialization sequence. **RTEMS** assumes that the board support package successfully completed its initialization activities. The **init_exec** directive completes the initialization sequence by performing the following actions:

- *Initializing internal* **RTEMS** *variables;*
- *Allocating system resources;*
- *Creating and starting the* ***System Initialization Task****;*
- *Creating and starting the* ***Idle Task****;*
- *Creating and starting the user initialization task(s); and*
- *Initiating multitasking.*

This directive **MUST** be called before any other **RTEMS** directives. The effect of calling any **RTEMS** directives before **init_exec** is unpredictable. Many of **RTEMS** actions during initialization are based upon the contents of the **Configuration Table** and **CPU Dependent** Information Table. For more information regarding the format and contents of these tables, please refer to the chapter **Configuring a System.**

The final step in the initialization sequence is the initiation of multitasking. When the scheduler and dispatcher are enabled, the highest priority, ready task will be dispatched to run. Control will not be returned to the board support package after multitasking is enabled.

D. Directives

This section details initialization manger's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

INIT_EXEC – Initialize RTEMS

CALLING SEQUENCE:

void init_exec (conftbl, cputbl)

INPUT:

config_table *conftbl;/* configuration table pointer */

struct cpu_info *cputbl; /* CPU information table pointer */

OUTPUT: NONE

DIRECTIVE STATUS CODES: NONE

DESCRIPTION:

This directive is called when the board support package has completed its initialization to allow **RTEMS** to initialize the application environment based upon the information in the **Configuration Table** and **CPU Dependent Information Table**.

NOTES:

This directive **MUST** be the first **RTEMS** directive called and it **DOES NOT RETURN** to the caller.

On some processors **RTEMS** does not require any information which is processor dependent. In this case, the **cpu_tbl** argument should be **NULL_CPU_TABLE**.

This directive causes all nodes in the system to verify that certain configuration parameters are the same as those of the local node. If an inconsistency is detected, then a fatal error is generated.

IV. TASK MANAGER

A. Introduction

The **task manager** provides a comprehensive set of directives to create, delete, and administer tasks. The directives provided by the task manager are:

Name	Directive Description
t_create	Create a task
t_ident	Get ID of a task
t_start	Start a task
t_restart	Restart a task
t_delete	Delete a task
t_suspend	Suspend a task
t_resume	Resume a task
t_setpri	Set task priority
t_mode	Change current task's mode
t_getnote	Get task notepad entry
t_setnote	Set task notepad entry

B. Background

1. Task Definition

Many definitions of a task have been proposed in computer literature. Unfortunately, none of these definitions encompasses all facets of the concept in a manner which is operating system independent. Several of the more common definitions are provided to enable each user to select a definition which best matches their own experience and understanding of the task concept:

- *the "dispatchable" unit.*
- *the entity to which the processor is allocated.*
- *the atomic unit of a real–time, multiprocessor system.*
- *single threads of execution which concurrently compete for resources.*
- *a sequence of closely related computations which can execute concurrently with other computational sequences.*

From **RTEMS'** perspective, a task is the smallest thread of execution which can compete on its own for system resources. A task is manifested by the existence of a Task Control Block (TCB).

2. Task Control Block

The TCB is an **RTEMS** defined data structure which contains all the information that is pertinent to the execution of a task. During system initialization, **RTEMS** reserves a TCB for each task configured. A TCB is allocated upon creation of the task and is returned to the TCB free list upon deletion of the task.

The TCB's elements are modified as a result of system calls made by the application in response to external and internal stimuli. TCBs are the only **RTEMS** internal data structure that can be accessed by an application via user extension routines. The TCB contains a task's name, ID, current priority, current and starting states, execution mode, set of notepad locations, TCB user extension pointer, scheduling control structures, as well as data required by a blocked task.

A task's context is stored in the TCB when a task switch occurs. When the task regains control of the processor, its context is restored from the TCB. When a task is restarted, the initial state of the task is restored from the starting context area in the task's TCB.

3. Task States

A task may exist in one of the following five states:

Task State	Description
executing	**Currently scheduled to the CPU**
ready	**May be scheduled to the CPU**
blocked	**Unable to be scheduled to the CPU**
dormant	**Created task that is not started**
non–existent	**Uncreated or deleted task**

Figure 4. Task States

An active task may occupy the executing, ready, blocked, or dormant state, otherwise the task is considered non–existent. One or more tasks may be active in the system simultaneously. Multiple tasks communicate, synchronize, and compete for system resources with each other via system calls. The multiple tasks appear to execute in parallel, but actually each is dispatched to the CPU for periods of time determined by the **RTEMS** scheduling algorithm. The scheduling of a task is based on its current state and priority.

4. Task Priority

A task's priority determines its importance in relation to the other tasks executing on the same processor. **RTEMS** supports 255 levels of priority ranging from 1 to 255. Tasks of numerically smaller priority values are more important tasks than tasks of numerically larger priority values. For example, a task at priority level 5 is of higher privilege than a task at priority level 10. There is no limit to the number of tasks assigned to the same priority.

Each task has a priority associated with it at all times. The initial value of this priority is assigned at task creation time. The priority of a task may be changed at any subsequent time.

Priorities are used by the scheduler to determine which ready task will be allowed to execute. In general, the higher the priority of a task, the more likely it is to receive processor execution time.

5. Task Mode

A task's mode is a combination of the following four components:

- *preemption*
- *ASR processsng*
- *timeslicing*
- *interrupt level*

It is used to modifying **RTEMS'** scheduling process and to alter the execution environment of the task.

The preemption component allows a task to determine when control of the processor is relinquished. If preemption is disabled (**NOPREEMPT**), the task will retain control of the processor as long as it is in the ready state – – even if a higher priority task is made ready. If preemption is enabled (**PREEMPT**) and a higher priority task is made ready, then the processor will be taken away from the current task immediately and given to t' higher priority task.

The timeslicing component is used by the **RTEMS** scheduler to determine how the processor is allocated to tasks of equal priority. If timeslicing is enabled (**TSLICE**), then **RTEMS** will limit the amount of time the task can execute before the processor is allocated to another ready task of equal priority. The length of the timeslice is application dependent and specified in the **Configuration Table**. If timeslicing is disabled (**NOTSLICE**), then the task will be allowed to execute until a task of higher priority is made ready. If **NOPREEMPT** is selected, then the timeslicing component is ignored by the scheduler.

The asynchronous signal processing component is used to determine when received signals are to be processed by the task. If signal processing is enabled (**ASR**), then signals sent to the task will be processed the next time the task executes. If signal processing is disabled (**NOASR**), then all signals received by the task will remain posted until signal processing is enabled. This component affects only tasks which have established a routine to process asynchronous signals.

The interrupt level component is used to determine which interrupts will be enabled when the task is executing. **INTR(n)** specifies that the task will execute at interrupt level **n**.

CONSTANT	DESCRIPTION	DEFAULT
PREEMPT	enable preemption	*
NOPREEMPT	disable preemption	
NOTSLICE	disable timeslicing	*
TSLICE	enable timeslicing	
ASR	enable ASR processing	*
NOASR	disable ASR processing	
INTR(o)	enable all interrupts	*
INTR(n)	execute at interrupt level n	

Figure 5. Task Mode Constants

6. Accessing Task Arguments

All RTEMS tasks are invoked with a single argument which is specified when they are started or restarted. The argument is commonly used to communicate some startup information to the task. The simplest manner in which to define a task which accesses it argument is:

```
task user_task (arg)
unsigned32 arg;
```

Application tasks requiring more information may view the single argument as a pointer to a parameter block. A task utilizing the argument in this manner may be defined as follows:

```
task user_task (arg_ptr)
struct user_task_args *arg_ptr;
```

where the structure, **user_task_args** is an application defined entity.

7. Floating Point Considerations

Creating a task with the **FP** flag results in additional memory being allocated for the TCB to store the state of the numeric coprocessor during task switches. This additional memory is not allocated for **NOFP** tasks. Saving and restoring the context of a **FP** task takes longer than that of a **NOFP** task because of the relatively large amount of time required for the numeric coprocessor to save or restore its computational state.

Since RTEMS was designed specifically for embedded military applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. The state of the numeric coprocessor is only saved when an **FP** task is dispatched and that task was not the last task to utilize the coprocessor. In a system with only one FP task, the state of the numeric coprocessor will never be saved or restored.

Although the overhead imposed by **FP** tasks is minimal, some applications may wish to completely avoid the overhead associated with **FP** tasks and still utilize a numeric

coprocessor. By preventing a task from being preempted while performing a sequence of floating point operations, a **NOFP** task can utilize the numeric coprocessor without incurring the overhead of a **FP** context switch. However, if this approach is taken by the application designer, **NO** tasks should be created as **FP** tasks.

If the supported processor type does not have hardware floating capabilities or a standard numeric coprocessor, **RTEMS** will not provide built–in support for hardware floating point on that processor. In this case, all tasks are considered **NOFP** whether created as a **FP** or **NOFP** task. A floating point emulation software library must be utilized for floating point operations.

8. Building an Attribute Set, Mode, or Mask

In general, an attribute set, mode, or mask is built by a bitwise OR of the desired options. The set of valid options is provided in the description of the appropriate directive. An option listed as a default is not required to appear in the option OR list, although it is a good programing practice to specify default options. If all defaults are desired, the option **DEFAULTS** should be specified on this call.

This example demonstrates the **attr** parameter needed to create a local task which utilizes the numeric coprocessor. The **attr** parameter could be **FP** or **LOCAL | FP**. The **attr** parameter can be set to **FP** because **LOCAL** is the default for all created tasks. If the task were global and used the numeric coprocessor, then the **attr** parameter would be **GLOBAL | FP**.

The following example will demonstrate the **mode** and **mask** parameters used with the **t_mode** directive to place a task at interrupt level 3 and make it non–preemptible. The **mode** should be set to **INTR(3) | NOPREEMPT** to indicate the desired preemption mode and interrupt level, while the **mask** parameter should be set to **INTRMODE | PREEMPTMODE** to indicate that the calling task's interrupt level and preemption mode are being altered.

C. Operations

1. Creating Tasks

The **t_create** directive creates a task by allocating a task control block, assigning the task a user–specified name, allocating it a stack and floating point context area, setting a user–specified initial priority, and assigning it a task ID. Newly created tasks are initially placed in the dormant state. All **RTEMS** tasks execute in the most privileged mode of the processor.

2. Obtaining Task IDs

When a task is created, **RTEMS** generates a unique task ID and assigns it to the created task until it is deleted. The task ID may be obtained by either of two methods. First, as the result of an invocation of the **t_create** directive, the task ID is stored in a user provided location. Second, the task ID may be obtained later using the **t_ident** directive. The task ID is used by other directives to manipulate this task.

3. Starting and Restarting Tasks

The **t_start** directive is used to place a dormant task in the **ready** state. This enables the task to compete, based on its current priority, for the processor and other system

resources. Any actions, such as suspension or change of priority, performed on a task prior to starting it are nullified when the task is started.

With the **t_start** directive the user specifies the task's starting address, initial execution mode, and argument. The argument is used to communicate some startup information to the task. As part of this directive, **RTEMS** initializes the task's stack based upon the task's initial execution mode and start address. The starting argument is passed to the task in accordance with the target processor's calling convention.

The **t_restart** directive restarts a task at its initial starting address with its original priority and execution mode, but with a possibly different argument. The new argument may be used to distinguish between the original invocation of the task and subsequent invocations. The task's stack and control block are modified to reflect their original creation values. Although references to resources that have been requested are cleared, resources allocated by the task are **NOT** automatically returned to **RTEMS**. A task cannot be restarted unless it has previously been started (i.e. dormant tasks cannot be restarted). All restarted tasks are placed in the **ready** state.

4. Suspending and Resuming Tasks

The **t_suspend** directive is used to place either the caller or another task into a suspended state. The task remains suspended until a **t_resume** directive is issued. This implies that a task may be suspended as well as blocked waiting either to acquire a resource or for the expiration of a timer.

The **t_resume** directive is used to remove another task from the suspended state. If the task is not also blocked, resuming it will place it in the ready state, allowing it to once again compete for the processor and resources. If the task was blocked as well as suspended, this directive clears the suspension and leaves the task in the blocked state.

5 Changing Task Priority

The **t_setpri** directive is used to obtain or change the current priority of either the calling task or another task. If the new priority requested is **CURRENT** or the task's actual priority, then the current priority will be returned and the task's priority will remain unchanged. If the task's priority is altered, then the task will be scheduled according to its new priority.

The **t_restart** directive resets the priority of a task to its original value.

6. Changing Task Mode

The **t_mode** directive is used to obtain or change the current execution mode of the calling task. A task's execution mode is used to enable preemption, timeslicing, ASR processing, and to set the task's interrupt level.

The **t_restart** directive resets the mode of a task to its original value.

7. Notepad Locations

RTEMS provides sixteen notepad locations for each task. Each notepad location may contain a **note** consisting of four bytes of information. **RTEMS** provides two directives, **t_setnote** and **t_getnote**, that enable a user to access and change the notepad locations. The **t_setnote** directive enables the user to set a task's notepad entry to a specified note.

The **t_getnote** directive allows the user to obtain the note contained in any one of the sixteen notepads of a specified task.

8. Task Deletion

RTEMS provides the **t_delete** directive to allow a task to delete itself or any other task. This directive removes all **RTEMS** references to the task, frees the task's control block, removes it from resource wait queues, and deallocates its stack as well as the optional floating point context. The task's name and ID become inactive at this time, and any subsequent references to either of them is invalid. In fact, **RTEMS** may reuse the task ID for another task which is created later in the application.

Unexpired delay timers (i.e., those used by **tm_wkafter** and **tm_wkwhen**) and timeout timers associated with the task are automatically deleted, however, other resources dynamically allocated by the task are **NOT** automatically returned to **RTEMS**. Therefore, before a task is deleted, all of its dynamically allocated resources should be deallocated by the user. This may be accomplished by instructing the task to delete itself rather than directly deleting the task. Other tasks may instruct a task to delete itself by sending a "delete self" message, event, or signal, or by restarting the task with special arguments which instruct the task to delete itself.

D. Directives

This section details the task manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **T_CREATE – Create a task**

CALLING SEQUENCE:

dir_status t_create (name, priority, stksize, mode, attr, & tid)

INPUT:

```
obj_name      name;     /* user–defined name      */
task_pri      priority; /* task priority          */
unsigned32    stksize;  /* stack size in bytes    */
unsigned32    mode;     /* initial task mode      */
unsigned32    attr;     /* task attributes        */
```

OUTPUT:

```
obj_id        *tid;     /* id of created task     */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task created successfully
E_SIZE	stack too small
E_MEMORY	no memory for stack segment
E_PRIORITY	invalid task priority
E_NOMP	multiprocessing not configured
E_TOOMANY	too many tasks created, or too many global objects

DESCRIPTION:

This directive creates a task which resides on the local node. It allocates and initializes a TCB, a stack, and an optional floating point context area. The **mode** parameter contains values which sets the task's initial execution mode. The **FP** attribute should be specified if the created task is to use a numeric coprocessor. For performance reasons, it is recommended that tasks not using the numeric coprocessor should specify the **NOFP** attribute. If the **GLOBAL** attribute is specified, the task can be accessed from remote nodes. The task id, returned in **tid**, is used in other task related directives to access the task. When created, a task is placed in the dormant state and can only be made ready to execute using the directive **t_start**.

NOTES:

This directive will not cause the calling task to be preempted.

Valid task priorities range from a high of 1 to a low of 255.

RTEMS supports a maximum of 256 interrupt levels which are mapped onto the interrupt levels actually supported by the target processor.

The requested stack size should be at least **MIN_STK_SIZE** bytes. The value of **MIN_STK_SIZE** is processor dependent. Application developers should consider the stack usage of the device drivers when calculating the stack size required for tasks which utilize the driver.

The following task attribute constants are defined by **RTEMS:**

CONSTANT	DESCRIPTION	DEFAULT
NOFP	does not use coprocessor	*
FP	uses numeric coprocessor	
LOCAL	local task	*
GLOBAL	global task	

The following task mode constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
PREEMPT	enable preemption	*
NOPREEMPT	disable preemption	
NOTSLICE	disable timeslicing	*
TSLICE	enable timeslicing	
ASR	enable ASR processing	*
NOASR	disable ASR processing	
INTR(0)	enable all interrupts	*
INTR(n)	execute at interrupt level n	

Tasks should not be made global unless remote tasks must interact with them. This avoids the system overhead incurred by the creation of a global task. When a global task is created, the task's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including tasks, is limited by the **num_gobjects** field in the **Configuration Table**.

2. T_IDENT – Get ID of a task

CALLING SEQUENCE:

dir_status t_ident (name, node, & tid)

INPUT:

```
obj_name      name;    /* user-defined task name    */
unsigned32    node;    /* nodes to be searched      */
```

OUTPUT:

```
obj_id        *tid;    /* task id returned          */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task identified successfully
E_NAME	invalid task name
E_NODE	invalid node id

DESCRIPTION:

This directive obtains the task id associated with the task name specified in **name**. A task may obtain its own id specifying **SELF** or its own task name in **name**. If the task name is not unique, then the task id returned will match one of the tasks with that name. However, this task id is not guaranteed to correspond to the desired task. The task id, returned in tid, is used in other task related directives to access the task.

NOTES:

This directive will not cause the running task to be preempted.

If **node** is **ALL_NODES**, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If **node** is a valid node number which does not represent the local node, then only the tasks exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

3. T_START – Start a task

CALLING SEQUENCE:

dir_status t_start (tid, saddr, arg)

INPUT:

```
obj_id      tid;     /* task id           */
task_ptr    saddr;   /* task entry point  */
unsigned32  arg;     /* argument          */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task started successfully
E_ID	invalid task id
ESTATE	task not in the dormant state
E_REMOTE	cannot start remote task

DESCRIPTION:

This directive readies the task, specified by **tid**, for execution based on the priority and execution mode specified when the task was created. The starting address of the task is given in **saddr**. The task's starting argument is contained in **arg**. This argument can be a single value or the address of a parameter block.

NOTES:

The calling task will be preempted if its preemption mode is enabled and the task being started has a higher priority.

Any actions performed on a **dormant** task such as suspension or change of priority are nullified when the task is initiated via the **t_start** directive.

4. T_RESTART – Restart a task

CALLING SEQUENCE:

dir_status t_restart (tid, arg)

INPUT:

```
obj_id      tid;    /* task id     */
unsigned32  arg;    /* argument    */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task restarted successfully
E_ID	task id invalid
E_STATE	task never started
E_REMOTE	cannot restart remote task

DESCRIPTION:

This directive resets the task specified by **tid** to begin execution at its original starting address. The task's priority and execution mode are set to the original creation values. If the task is currently blocked, **RTEMS** automatically makes the task ready. A task can be restarted from any state, except the dormant state.

The task's starting argument is contained in **arg**. This argument can be a single value or the address of a parameter block. This new argument may be used to distinguish between the initial **t_start** of the task and any ensuing calls to **t_restart** of the task. This can be beneficial in deleting a task. Instead of deleting a task using the **t_delete** directive, a task can delete another task by restarting that task, and allowing that task to release resources back to **RTEMS** and then delete itself.

NOTES:

If **tid** is **SELF**, the calling task will be restarted and will not return from this directive.

The calling task will be preempted if its preemption mode is enabled and the task being restarted has a higher priority.

The task must reside on the local node, even if the task was created with the **GLOBAL** option.

5. **T_DELETE – Delete a task**

CALLING SEQUENCE:

dir_status t_delete (tid)

INPUT:

obj_id tid; /* task id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task deleted successfully
E_ID	invalid task id
E_REMOTE	cannot delete remote task

DESCRIPTION:

This directive deletes a task, either the calling task or another task, as specified by **tid**. **RTEMS** stops the execution of the task and reclaims the stack memory, any allocated delay or timeout timers, and the TCB. **RTEMS** does not reclaim region segments, partition buffers, semaphores, event timers, or rate monotonic timers.

NOTES:

A task is responsible for releasing its resources back to **RTEMS** before deletion. To insure proper deallocation of resources, a task should not be deleted unless it is unable to execute or does not hold any **RTEMS** resources. If a task holds **RTEMS** resources, the task should be allowed to deallocate its resources before deletion. A task can be directed to release its resources and delete itself by restarting it with a special argument or by sending it a message, an event, or a signal.

Deletion of the current task (**SELF**) will force **RTEMS** to select a another task to execute.

When a global task is deleted, the task id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The task must reside on the local node, even if the task was created with the **GLOBAL** option.

6. T_SUSPEND – Suspend a task

CALLING SEQUENCE:

dir_status t_suspend (tid)

INPUT:

obj_id tid; /* task id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task suspended successfully
E_ID	invalid task id
E_ALREADY	task already suspended

DESCRIPTION:

This directive suspends the task specified by **tid** from further execution by placing it in the suspended state. This state is additive to any other blocked state that the task may already be in. The task will not execute again until another task issues the **t_resume** directive for this task and any blocked state has been removed.

NOTES:

The requesting task can suspend itself by specifying **SELF** as **tid**. In this case, the task will be suspended and a successful return code will be returned when the task is resumed.

Suspending a global task which does not reside on the local node will generate a request to the remote node to suspend the specified task.

7. T_RESUME – Resume a task

CALLING SEQUENCE:

dir_status t_resume (tid)

INPUT:

obj_id tid; /* task id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task resumed successfully
E_ID	invalid task id
E_STATE	task not suspended

DESCRIPTION:

This directive removes the task specified by **tid** from the suspended state. If the task is in the ready state after the suspension is removed, then it will be scheduled to run. If the task is still in a blocked state after the suspension is removed, then it will remain in that blocked state.

NOTES:

The running task may be preempted if its preemption mode is enabled and the local task being resumed has a higher priority.

Resuming a global task which does not reside on the local node will generate a request to the remote node to resume the specified task.

8. T_SETPRI – Set task priority

CALLING SEQUENCE:

dir_status t_setpri (tid, priority, & ppriority)

INPUT:

```
obj_id    tid;          /* task id          /*
task_pri  priority;     /* new priority     /*
```

OUTPUT:

```
task_pri  *ppriority;   /* previous priority */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	task priority set successfully
E_ID	invalid task id
E_PRIORITY	invalid task priority

DESCRIPTION:

This directive manipulates the priority of the task specified by **tid.** A **tid** of **SELF** is used to indicate the calling task. When **priority** is not equal to **CURRENT**, the specified task's previous priority is returned in **ppriority**. When **priority** is **CURRENT**, the specified task's current priority is returned in **ppriority**. Valid priorities range from a high of 1 to a low of 255.

NOTES:

The calling task may be preempted if its preemption mode is enabled and it lowers its own priority or raises another task's priority.

Setting the priority of a global task which does not reside on the local node will generate a request to the remote node to change the priority of the specified task.

9. T_MODE – Change current task's mode

CALLING SEQUENCE:

dir_status t_mode (mode, mask, & pmode)

INPUT:

```
unsigned32    mode;    /* new mode values       /*
unsigned32    mask;    /* mode fields to change */
```

OUTPUT:

```
unsigned32    *pmode;  /* previous mode         /*
```

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive manipulates the execution mode of the calling task. A task's execution mode enables and disables preemption, timeslicing, asynchronous signal processing, as well as specifying the current interrupt level. To modify an execution mode, the mode class(es) to be changed must be specified in the **mask** parameter and the desired mode(s) must be specified in the **mode** parameter.

NOTES:

The calling task will be preempted if it enables preemption and a higher priority task is ready to run.

Enabling timeslicing has no effect if preemption is enabled.

A task can obtain its current execution mode, without modifying it, by calling this directive with a **mask** value of **CURRENT**.

To temporarily disable the processing of a valid ASR, a task should call this directive with the **NOASR** indicator specified in **mode**.

The following task mode constants are defined by **RTEMS:**

CONSTANT	DESCRIPTION	DEFAULT
PREEMPT	enable preemption	*
NOPREEMPT	disable preemption	
NOTSLICE	disable timeslicing	*
TSLICE	enable timeslicing	
ASR	enable ASR processing	*
NOASR	disable ASR processing	
INTR(0)	enable all interrupts	*
INTR(n)	execute at interrupt level n,	

The following mask constants are defined by **RTEMS:**

CONSTANT	DESCRIPTION
CURRENT	obtain current mode
PREEMPTMODE	select preemption mode
TSLICEMODE	select timeslicing mode
ASRMODE	select ASR processing mode
INTRMODE	select interrupt level

10. T_GETNOTE – Get task notepad entry

CALLING SEQUENCE:

dir_status t_getnote (tid, notepad, & note)

INPUT:

```
obj_id      tid;      /* task id          */
unsigned32  notepad;  /* notepad location */
```

OUTPUT:

```
unsigned32  *note;    /* note value       */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	note obtained successfully
E_ID	invalid task id
E_NUMBER	invalid notepad location

DESCRIPTION:

This directive returns the note contained in the **notepad** location of the task specified by **tid**.

NOTES:

This directive will not cause the running task to be preempted.

If **tid** is set to **SELF**, the calling task accesses its own notepad.

The sixteen notepad locations can be accessed using the constants **NOTEPAD_0** through **NOTEPAD_15.**

Getting a note of a global task which does not reside on the local node will generate a request to the remote node to obtain the notepad entry of the specified task.

11. T_SETNOTE – Set task notepad entry

CALLING SEQUENCE:

dir_status t_setnote (tid, notepad, note)

INPUT:

```
obj_id      tid;     /* task id              */
unsigned32  notepad; /* notepad location     */
unsigned32  note;    /* new value for note   */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	task's note set successfully
E_ID	invalid task id
E_NUMBER	invalid notepad location

DESCRIPTION:

This directive sets the **notepad** entry for the task specified by **tid** to the value note.

NOTES:

If **tid** is set to **SELF**, the calling task accesses its own notepad locations.

This directive will not cause the running task to be preempted.

The sixteen notepad locations can be accessed using the constants **NOTEPAD_0** through **NOTEPAD_15.**

Setting a notepad location of a global task which does not reside on the local node will generate a request to the remote node to set the specified notepad entry.

V. INTERRUPT MANAGER

A. Introduction

Any real–time executive must provide a mechanism for quick response to externally generated interrupts to satisfy the critical time constraints of the application. The **interrupt manager** provides this mechanism for **RTEMS.** This manager permits quick interrupt response times by providing the critical ability to alter task execution which allows a task to be preempted upon exit from an ISR The interrupt manager includes the following directive:

Name	Directive Description
i_catch	Establish an ISR

B. Background

1. Processing an Interrupt

The interrupt manager allows the application to connect a C routine to a hardware interrupt vector. When an interrupt occurs, the processor will automatically vector to **RTEMS. RTEMS** saves and restores all registers which are not preserved by the normal C calling convention for the target processor and invokes the user's ISR The user's ISR is responsible for processing the interrupt, clearing the interrupt if necessary, and device specific manipulation.

The **i_catch** directive connects a procedure to an interrupt vector. The inter*rupt service routine is assumed* to abide by these conventions and have the following C calling sequence:

```
void isr ( vector )
unsigned32 vector;        /* vector number    /*
```

The **vector** number argument is provided by **RTEMS** to allow the application to identify the interrupt source. This could be used to allow a single routine to service interrupts from multiple instances of the same device. For example, a single routine could service interrupts from multiple serial ports and use the vector number to identify which port requires servicing.

To minimize the masking of lower or equal priority level interrupts, the ISR should perform the minimum actions required to service the interrupt. Other non–essential actions should be handled by application tasks. Once the user's ISR has completed, it returns control to the **RTEMS** interrupt manager which will perform task dispatching and restore the registers saved before the ISR was invoked.

This guarantees that proper task scheduling and dispatching are performed at the conclusion of an ISR. A system call made by the ISR may have readied a task of higher priority than the interrupted task. Therefore, when the ISR completes, the postponed dispatch processing must be performed. No dispatch processing is performed as part of directives which have been invoked by an ISR.

Applications must adhere to the following rule if proper task scheduling and dispatching is to be performed:

The interrupt manager must be used for all ISRs which may be interrupted by the highest priority ISR which involves an **RTEMS** *directive.*

Consider a processor which allows a numerically low interrupt level to interrupt a numerically greater interrupt level. In this example, if an RTEMS directive is used in a level 4 ISR, then all ISRs which execute at levels 0 through 4 must use the interrupt manager.

Interrupts are nested whenever an interrupt occurs during the execution of another ISR. **RTEMS** supports efficient interrupt nesting by allowing the nested ISRs to terminate without performing any dispatch processing. Only when the outermost ISR terminates will the postponed dispatching occur.

2. RTEMS Interrupt Levels

Many processors support multiple interrupt levels or priorities. The exact number of interrupt levels is processor dependent. **RTEMS** internally supports 256 interrupt levels which are mapped to the processor's interrupt levels. For specific information on the mapping between **RTEMS** and the target processor's interrupt levels, refer to the **Interrupt Processing** chapter of the **C Applications Supplement** document for a specific target processor.

3. Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, **RTEMS** disables all maskable interrupts before the execution of the section and restores them to the previous level upon completion of the section. **RTEMS** has been optimized to insure that interrupts are disabled for a minimum length of timer. The maximum length of time interrupts are disabled by **RTEMS** is processor dependent and is detailed in the **Timing Specification** chapter of the **C Applications Supplement** document for a specific target processor.

Non–Maskable Interrupts (NMI) cannot be disabled, and ISRs which execute at this level **MUST NEVER** issue **RTEMS** system calls. If a directive is invoked, unpredictable results may occur due to the inability of **RTEMS'** to protect its critical sections. However, ISRs that make no system calls may safely execute as non–maskable interrupts.

C. Operations

1. Establishing an ISR

The **i_catch** directive establishes an ISR for the system. The address of the ISR and its associated CPU vector number are specified to this directive. This directive installs the **RTEMS** interrupt wrapper in the processor's Interrupt Vector Table and the address of the user's ISR in the **RTEMS'** Vector Table. This directive returns the previous contents of the specified vector in the **RTEMS'** Vector Table.

2. Directives Allowed from an ISR

Using the interrupt manager insures that **RTEMS** knows when a directive is being called from an ISR. The ISR may then use system calls to synchronize itself with an application task. The synchronization may involve messages, events or signals being passed by the ISR to the desired task. Directives invoked by an ISR must operate only on objects which

reside on the local node. The following is a list of **RTEMS** system calls that may be made from an ISR:

- *Task Management*
  t_getnote, t_setnote, t_suspend, t_resume
- *Message Event, and Signal Management*
  q_send, q_urgent

  ev_send

  as_send
- *Semaphore Management*
  sm_v
- *Time Management*
  tm_get, tm_tick
- *Dual–Ported Memory Management*
  dp_ext2int, dp_int2ext
- *Fatal Error Management*
  k_fatal
- *Multiprocessing*
  mp_announce

D. Directives

This section details the interrupt manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

I_CATCH – Establish an ISR

CALLING SEQUENCE:

void i_catch (israddr, vector, & oldisr)

INPUT:

```
proc_ptr      israddr;  /* ISR entry point            */
unsigned32    vector;   /* interrupt vector number    */
```

OUTPUT:

```
proc_ptr      *oldisr;  /* previous ISR entry point    */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	ISR established successfully
E_NUMBER	illegal vector number
E_ADDRESS	illegal ISR entry point

DESCRIPTION:

This directive establishes an Interrupt Service Routine (ISR) for the specified interrupt **vector** number. The **israddr** parameter specifies the entry point of the ISR The entry point of the previous ISR for the specified vector is returned in oldisr.

NOTES:

This directive will not cause the calling task to be preempted.

VI. TIME MANAGER

A. Introduction

The **time manager** provides support for both clock and timer facilities. The directives provided by the time manager are:

Name	Directive Description
tm_set	Set system date and time
tm_get	Get system date and time
tm_wkafter	Wake up after interval
tm_wkwhen	Wake up when specified
tm_evafter	Send event set after interval
tm_evwhen	Send event set when specified
tm_evevery	Send periodic event set
tm_delete	Delete event timer
tm_tick	Announce a clock tick

B. Background

1. Required Support

For the features provided by the time manager to be utilized, periodic timer interrupts are required. Therefore, a real–time clock or some kind of hardware timer is necessary to create the timer interrupts. The **tm_tick** directive is normally called by the timer ISR to announce to RTEMS that a system clock tick has transpired. Elapsed time is measured in ticks. A tick is defined to be a integral number of milliseconds which is specified by the user in the **Configuration Table.**

2. Time and Date Data Structure

The clock facilities of the time manager operate upon a calendar time. These directives utilize the following date and time structure:

```
struct time_info {
    unsigned16    year;/*     year A.D. ; greater than 1987     */
    unsigned8     month;      /*month, 1 – 12                    */
    unsigned8     day;        /* day, 1 – 31                     */
    unsigned16    hour;       /* hour, 0 – 23                    */
    unsigned8     minute;     /* minute, 0 – 59                  */
    unsigned8     second;     /* second, 0 – 59                  */
    unsigned32    ticks;      /* elapsed ticks between seconds   */
};
```

3. Timer Types

The following types of timers are maintained by the time manager:

- *sleep timers*
- *event timers*
- *timeouts*

A **sleep timer** allows a task to delay for a given interval or up until a given time, ad then wake and continue execution. This type of timer is created automatically by the **tm_wkafter** and **tm_wkwhen** directives and, as a result, does not have a **RTEMS** ID. Once activated, a sleep timer cannot be explicitly deleted. Each task may activate one and only one sleep timer at a time.

The **event timer** allows a task to send a event set to itself either after a given interval or at a given time. This type of timer is created automatically by the **tm_evafter**, **tm_evevery**, and **tm_evwhen** directives. All event timers are assigned a unique **RTEMS** ID which can be used to delete the timer. A task can have multiple event timers active simultaneously. The task must use the **ev_receive** directive to determine if the events have been posted.

Timeouts are a special type of timer automatically created when the timeout option is used on the **q_receive**, **en_receive**, **sm_p** and **rn_getseg** directives. Each task may have one and only one timeout active at a time. When a timeout expires, it unblocks the task with a timeout status code.

Timers affect only the calling task, either by putting it to sleep or sending it an event set. For any particular task, multiple event timers can be used in combination with a single timeout or sleep timer. Under no circumstances, can a task have both a sleep timer and a timeout active simultaneously.

A **Timer Control Block** (TMCB) is allocated as part of creating a timer. The TMCB is used by **RTEMS** to manage the timer. TMCBs are automatically freed when the timer expires.

4. Timeslicing

Timeslicing is a task scheduling discipline in which tasks of equal priority are executed for a specific period of time before control of the CPU is passed to another task. It is also sometimes referred to as the **automatic round–robin** scheduling algorithm. The length of time allocated to each task is known as the **quantum** or **timeslice.**

The system's timeslice is defined as a integral number of ticks, and is specified in the **Configuration Table.** The timeslice is defined for the entire system of tasks, but timeslicing is enabled and disabled on a per task basis.

The **tm_tick** directive implements timeslicing by decrementing the running task's time–remaining counter when both timeslicing and preemption are enabled. If the task's timeslice has expired, then that task will be preempted if there exists a ready task of equal priority.

C. Operations

1. Announcing a Tick

RTEMS provides the **tm_tick** directive which is called from the user's real–time clock ISR to inform RTEMS that a tick has elapsed. The tick frequency value, defined in milliseconds, is a configuration parameter found in **RTEMS's Configuration Table**. RTEMS divides 1000 milliseconds (one second) by the number of milliseconds per tick to determine the number of calls to the **tm_tick** directive per second. The frequency of **tm_tick** calls determines the resolution (granularity) for all time dependent RTEMS actions. For example, calling **tm_tick** ten times per second yields a higher resolution than calling **tm_tick** two times per second. The **tm_tick** directive is responsible for maintaining both calendar time and the dynamic set of timers.

2. Setting and Obtaining the Time

The **tm_set** directive allows a task or a ISR to set the date and time maintained by RTEMS. Calendar time operations will return a error code if invoked before the date and time have been set. The **tm_get** directive allows a task or a ISR to obtain the current date and time.

3. Using a Sleep Timer

The **tm_wkafter** directive creates a sleep timer which allows a task to go to sleep for a specified interval. The task is blocked until the delay interval has elapsed, at which time the task is unblocked. A task calling the **tm_wkafter** directive with a delay interval of **YIELD** ticks will yield the processor to any other ready task of equal or greater priority and remain ready to execute.

The **tm_wkwhen** directive creates a sleep timer which allows a task to go to sleep until a specified date and time. The calling task is blocked until the specified date and time has occurred, at which time the task is unblocked.

4. Using an Event Timer

The **tm_evafter** directive creates an event timer which allows a task to be sent a specified event set after a specified interval. The **tm_evwhen** directive creates an event timer which is programmed to expire at a future date and time. The **tm_evevery** directive is similar to the **tm_evafter** directive except that the created timer is rearmed rather than automatically deleted at the end of the interval. This results in a event set being sent at regular intervals rather than just a single time.

All three directives return a unique timer ID generated by RTEMS to the calling task.

The calling task is not blocked by either the **tm_evafter**, **tm_evevery**, or the **tm_evwhen** directive and must use the **ev_receive** directive to obtain the event set.

5. Deleting a Timer

The **tm_delete** directive is used to delete an event timer. The timer's control block is returned to the TMCB free list when the event timer is deleted. The timers created by **tm_evevery** are not automatically deleted by RTEMS and must be explicitly deleted by the application. Other interval and event timers are automatically deleted upon expiration.

D. Directives

This section details the time manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **TM_SET – Set system date and time**

CALLING SEQUENCE:

dir_status tm_set (&timebuf)

INPUT:

time_buffer *timebuf; /* date/time pointer */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	date and time set successfully
E_CLOCK	invalid time buffer

DESCRIPTION:

This directive sets the system date and time. The date, time, and ticks in the **timebuf** structure are all range–checked, and an error is returned if any one is out of its valid range.

NOTES:

Years before 1988 are invalid.

The system date and time are based on the configured tick rate (number of milliseconds in a tick).

Setting the time forward may cause a higher priority task, blocked waiting on a specific time, to be made ready. In this case, the calling task will be preempted after the next clock tick.

Reinitializing **RTEMS** causes the system date and time to be reset to an uninitialized state. Another call to **tm_set** is required to re–initialize the system date and time to application specific specifications.

2. **TM_GET – Get system date and time**

CALLING SEQUENCE:

dir_status tm_get (&timebuf)

INPUT: NONE

OUTPUT:

time_buffer *timebuf; /* date/time pointer */

DIRECTIVE STATUS CODES:

SUCCESSFUL current time obtained successfully
E_NOTDEFINED system date and time is not set

DESCRIPTION:

This directive obtains the system date and time. If the date and time have not been set with a previous call to **tm_set**, then the **E_NOTDEFINED** status code is returned.

NOTES:

This directive is callable from an ISR.

This directive will not cause the running task to be preempted. Re–initializing **RTEMS** causes the system date and time to be reset to an uninitialized state. Another call to **tm_set** is required to re–initialize the system date and time to application specific specifications.

3. TM_WKAFTER – Wake up after Interval

CALLING SEQUENCE:

dir_status tm_wkafter (ticks)

INPUT:

interval ticks; /* number of ticks to wait */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive blocks the calling task for the specified number of system clock ticks. When the requested interval has elapsed, the task is made ready. The **tm_tick** directive automatically updates the delay period.

NOTES:

Setting the system date and time with the **tm_set** directive has no effect on a **tm_wkafter** blocked task.

A task may give up the processor and remain in the ready state by specifying a value of **YIELD** in ticks.

The maximum timer interval that can be specified is the maximum **value** which can be represented by the **unsigned32** type.

4. TM_WKWHEN – Wake up when specified

CALLING SEQUENCE:

dir_status tm_wkwhen (timebuf)

INPUT:

time_buffer *timebuf; /* date/time pointer */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	awakened at date/time successfully
E_CLOCK	invalid time buffer
E_NOTDEFINED	system date and time is not set

DESCRIPTION:

This directive blocks a task until the date and time specified in **timebuf**. At the requested date and time, the calling task will be unblocked and made ready to execute.

NOTES:

The ticks portion of the timebuf structure is ignored. The timing granularity of this directive is a second.

5. TM_EVAFTER – Send event set after Interval

CALLING SEQUENCE:

dir_status tm_evafter (ticks, event, & tmid)

INPUT:

```
interval      ticks;    /* ticks until event      /*
event_set     event;    /* event set              /*
```

OUTPUT:

```
obj_id        *tmid;    /* id assigned to timer   /*
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	event timer set up successfully
E_TOOMANY	too many timers allocated

DESCRIPTION:

This directive sets up a timer directing **RTEMS** to send the event set, **event**, to the calling task after **ticks** system clock ticks have elapsed. The id for the created timer is returned in **tmid.**

The calling task must call **en_receive** to receive these events and will block until the timer expires. The **tm_tick** directive automatically adjusts the delay period.

NOTES:

This directive will not cause the calling task to be preempted.

Setting the system date and time by way of the **tm_set** directive has no effect on the countdown of the timer.

The maximum timer interval that can be specified is the maximum **value** which can be represented by the **unsigned32** type.

6. TM_EVWHEN – Send event set when specified

CALLING SEQUENCE:

dir_status tm_evwhen (timebuf, event, & tmid)

INPUT:

```
time_buffer    *timebuf;     /* time/date pointer     /*
event_set      event;        /* event set             /*
```

OUTPUT:

```
obj_id         *tmid;        /* id assigned to timer  */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	event timer set up successfully
E_CLOCK	invalid time buffer
E_NOTDEFINED	system date and time is not set
E_TOOMANY	too many timers allocated

DESCRIPTION:

This directive sets up a timer directing **RTEMS** to send the event set, **event**, to the calling task at the date and time specified in **timebuf**. The id for the created timer is returned in **tmid**. The calling task must call the **en_receive** directive to receive these events.

NOTES:

The ticks portion of the **timebuf** structure is set to zero. The timing granularity of this directive is a second.

7. TM_EVEVERY – Send periodic event set

CALLING SEQUENCE:

dir_status tm_evevery (ticks, event, & tmid)

INPUT:

```
interval      ticks;    /* ticks between events   */
event_set     event;    /* event set              */
```

OUTPUT:

```
obj_id        *tmid;    /* id assigned to timer   */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	event timer set up successfully
E_TOOMANY	too many timers allocated
E_NUMBER	interval of zero is invalid

DESCRIPTION:

This directive sets up an interval timer directing **RTEMS** to send the event set, event, to the calling task after every occurrence of **ticks** system clock ticks. The id for the created timer is returned in **tmid**.

The calling task must call **ev_receive** to receive these events and will block until the timer expires. The **tm_tick** directive is used to determine the period between each sending of the event set.

NOTES:

The directive **tm_delete** must be used to stop the timer from sending the event set.

This directive will not cause the calling task to be preempted.

Setting the system date and time by way of the **tm_set** directive has no effect on the created timer.

The maximum timer interval that can be specified is the maximum **value** which can be represented by the **unsigned32** type.

8. TM_DELETE – Delete event timer

CALLING SEQUENCE:

dir_status tm_delete (tmid)

INPUT:

obj_id tmid; /* timer id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	event timer deleted successfully
E_ID	invalid timer id

DESCRIPTION:

This directive deletes the event timer specified by tmid. This event timer was scheduled by the **tm_evafter**, the **tm_evwhen**, or the **tm_evevery** directives.

NOTES:

This directive will not cause the calling task to be preempted.

9. TM_TICK – Announce a clock tick

CALLING SEQUENCE:

dir_status tm_tick ( )

INPUT: NONE

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive announces to **RTEMS** that a system clock tick has occurred. The directive is usually called from the timer interrupt **ISR** of the local processor. This directive maintains the system date and time, decrements timers for delayed tasks, timeouts, event timers, rate monotonic timers, and implements timeslicing.

NOTES:

This directive is typically called from an **ISR**.

The **ms_tick** and **tslice** parameters in the **Configuration Table** contain the number of milliseconds per tick and number of ticks per timeslice, respectively.

VII. SEMAPHORE MANAGER

A. Introduction

The **semaphore manager** utilizes standard Dijkstra counting semaphores to provide synchronization and mutual exclusion capabilities. The directives provided by the semaphore manager are:

Name	Directive Description
sm_create	Create a semaphore
sm_ident	Get ID of a semaphore
sm_delete	Delete a semaphore
sm_p	Acquire a semaphore
sm_v	Release a semaphore

B. Background

A semaphore can be viewed as a protected variable whose value can be modified only with the **sm_create**, **sm_p**, and **sm_v** directives. **RTEMS** supports both binary and counting semaphores. A **binary semaphore** is restricted to values of zero or one, while a **counting semaphore** can assume any non–negative integer value.

A binary semaphore can be used to control access to a single resource. In particular, it can be used to enforce mutual exclusion for a critical section in user code. In this instance, the semaphore would be created with an initial count of one to indicate that no task is executing the critical section of code. Upon entry to the critical section, a task must issue the **sm_p** directive to prevent other tasks from entering the critical section. Upon exit from the critical section, the task must issue the **sm_v** directive to allow another task to execute the critical section.

A counting semaphore can be used to control access to a pool of two or more resources. For example, access to three printers could be administered by a semaphore created with an initial count of three. When a task requires access to one of the printers, it issues the **sm_p** directive to obtain access to a printer. If a printer is not currently available, the task can wait for a printer to become available or return immediately. When the task has completed printing, it should issue the **sm_v** directive to allow other tasks access to the printer.

Task synchronization may be achieved by creating a semaphore with an initial count of zero. One task waits for the arrival of another task by issuing a **sm_p** directive when it reaches a synchronization point. The other task performs a corresponding **sm_v** operation when it reaches its synchronization point, thus unblocking the pending task.

1. Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attributes. The set of valid attributes is provided in the description of the **sm_create** and **sm_p** directives. An attribute listed as a default is not required to appear in the attribute OR list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute **DEFAULTS** should be specified on this call.

This example demonstrates the **attr** parameter needed to create a local semaphore with a task priority waiting queue discipline. The **attr** parameter could be **PRIORITY** or **LOCAL | PRIORITY**. The **attr** parameter can be set to **PRIORITY** because **LOCAL** is the default for all created tasks. If a similar semaphore were to be known globally, then the **attr** parameter would be **GLOBAL | PRIORITY**.

C. Operations

1. Creating a Semaphore

The **sm_create** directive creates a semaphore with a user–specified name as well as an initial count. At create time the method for placing waiting tasks in the semaphore's task wait queue (FIFO or task priority) is specified. **RTEMS** allocates a **Semaphore Control Block** (SMCB) from the SMCB free list. This data structure is used by **RTEMS** to manage the newly created semaphore. Also, a unique semaphore ID is generated and returned to the calling task.

2. Obtaining Semaphore IDs

When a semaphore is created, **RTEMS** generates a unique semaphore ID and assigns it to the created semaphore until it is deleted. The semaphore ID may be obtained by either of two methods. First, as the result of an invocation of the **sm_create** directive, the semaphore ID is stored in a user provided location. Second, the semaphore ID may be obtained later using the **sm_indent** directive. The semaphore ID is used by other semaphore manager directives to access this semaphore.

3. Acquiring a Semaphore

The **sm_p** directive is used to acquire the specified semaphore. A simplified version of the **sm_p** directive can be described as follows:

```
if semaphore's count is greater than zero
     then decrement semaphore's count
     else wait for release of semaphore
return SUCCESSFUL
```

When the semaphore cannot be immediately acquired, one of the following situations applies:

- *By default, the calling task will wait forever to acquire the semaphore.*
- *Specifying* NOWAIT *forces an immediate return with an error status code.*
- *Specifying a timeout limits the interval task will wait before returnmg with an error status code.*

If the task waits to acquire the semaphore, then it is placed in the semaphore's task wait queue in either FIFO or task priority order. All tasks waiting on a semaphore are returned an error code when the message queue is deleted.

4. Releasing a Semaphore

The **sm_v** directive is used to release the specified semaphore. A simplified version of the **sm_v** directive can be described as follows:

```
if no tasks are waiting on this semaphore
     then increment semaphore's count
     else assign semaphore to a waiting task
return SUCCESSFUL
```

5. Semaphore Deletion

The **sm_delete** directive removes a semaphore from the system and frees its control block. A semaphore can be deleted by any task that knows the semaphore's ID. As a result of this directive, all tasks blocked waiting to acquire the semaphore will be readied and returned a status code which indicates that the semaphore was deleted. Any subsequent references to the semaphore's name and ID are invalid.

D. Directives

This section details the semaphore manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **SM_CREATE – Create a semaphore**

CALLING SEQUENCE:

dir_status sm_create (name, count, attr, & smid)

INPUT:

```
obj_name     name;    /* user–defined name    */
unsigned32   count;   /* initial count        */
unsigned32   attr;    /* attributes           */
```

OUTPUT:

```
obj_id       *smid;   /* smid assigned        /*
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore created successfully
E_TOOMANY	too many semaphores created
E_NOMP	multiprocessing not configured
E_TOOMANY	too many global objects

DESCRIPTION:

This directive creates a semaphore which resides on the local node. The created semaphore has the user–defined name specified in **name** and the initial count specified in count. For control and maintenance of the semaphore, RTEMS allocates and initializes a SMCB. The RTEMS–assigned semaphore id is returned in **smid**. This semaphore id is used with other semaphore related directives to access the semaphore.

Specifying **PRIORITY** in **attr** causes tasks waiting for a semaphore to be serviced according to task priority. When **FIFO** is selected, tasks are serviced in First In–First Out order.

NOTES:

This directive will not cause the calling task to be preempted.

The following semaphore attribute constants are defined by RTEMS:

CONSTANT	DESCRIPTION	DEFAULT
FIFO	tasks wait by FIFO	*
PRIORITY	tasks wait by priority	
NOLIMIT	unlimited queue size	*
LIMIT	limit queue size to count	
LOCAL	local semaphore	*
GLOBAL	global semaphore	

Semaphores should not be made global unless remote tasks must interact with the created semaphore. This is to avoid the system overhead incurred by the creation of a global semaphore. When a global semaphore is created, the semaphore's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including semaphores, is limited by the **num_gobjects** field in the **Configuration Table.**

2. SM_IDENT – Get ID of a semaphore

CALLING SEQUENCE:

dir_status sm_ident (name, node, & smid)

INPUT:

```
obj_name     name;   /* user–defined name    */
unsigned32   node;   /* node(s) to search    */
```

OUTPUT:

```
obj_id       *smid;  /* semaphore id         /*
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore identified successfully
E_NAME	semaphore name not found
E_NODE	invalid node id

DESCRIPTION:

This directive obtains the semaphore associated with the semaphore **name**. If the semaphore name is not unique, then the semaphore id will match one of the semaphores with that name. However, this semaphore id is not guaranteed to correspond to the desired semaphore. The semaphore id is used by other semaphore related directives to access the semaphore.

NOTES:

This directive will not cause the running task to be preempted.

If **node** is **ALL_NODES**, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If **node** is a valid node number which does not represent the local node, then only the semaphores exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

3. SM_DELETE – Delete a semaphore

CALLING SEQUENCE:

dir_status sm_delete (smid)

INPUT:

obj_id smid; /* semaphore id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore deleted successfully
E_ID	invalid semaphore id
E_REMOTE	cannot delete remote semaphore

DESCRIPTION:

This directive deletes the semaphore specified by **smid**. Any tasks that are waiting on this semaphore are unblocked with a status code for a deleted semaphore. The SMCB for this semaphore is reclaimed by **RTEMS.**

NOTES:

The calling task will be preempted if it enabled by the task's execution mode and a higher priority local task is waiting on the deleted semaphore. The calling task will **NOT** be preempted if all of the tasks that are waiting on the semaphore are remote tasks.

The calling task does not have to be the task that created the semaphore. Any local task that knows the semaphore id can delete the semaphore.

When a global semaphore is deleted, the semaphore id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The semaphore must reside on the local node, even if the semaphore was created with the **GLOBAL** option.

Proxies, used to represent remote tasks, are reclaimed when the semaphore is deleted.

4. **SM_P – Acquire a semaphore**

CALLING SEQUENCE:

dir_status sm_p (smid, options, timeout)

INPUT:

```
obj_id      smid;      /* semaphore id    */
unsigned32  options;   /* option set      */
interval    timeout;   /* wait interval   */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore obtained successfully
E_UNSATISFIED	semaphore not available
E_TIMEOUT	timed out waiting for semaphore
E_DELETE	semaphore deleted while waiting
E_ID	invalid semaphore id

DESCRIPTION:

This directive acquires the semaphore specified by **smid**. The **WAIT** and **NOWAIT** options of the options parameter are used to specify whether the calling task wants to wait for the semaphore to become available or return immediately. For either option, if the current semaphore count is positive, then it is decremented by one and the semaphore is successfully acquired by returning immediately with a successful return code.

If the calling task chooses to return immediately and the current semaphore count is zero or negative, then a status code indicating that the semaphore is not available is returned. If the calling task chooses to wait for a semaphore and the current semaphore count is zero or negative, then it is decremented by one and the calling task is placed on the semaphore's wait queue and blocked. If the semaphore was created with the **PRIORITY** option, then the calling task is inserted into the queue according to its priority. However, if the semaphore was created with the **FIFO** option, then the calling task is placed at the rear of the wait queue.

The timeout parameter specifies the maximum interval the calling task is willing to be blocked waiting for the semaphore. If it is set to **NOTIMEOUT**, then the calling task will wait forever.

NOTES:

The following semaphore acquisition option constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
WAIT	wait for semaphore	*
NOWAIT	do NOT wait for semaphore	

Attempting to obtain a global semaphore which does not reside on the local node will generate a request to the remote node to access the semaphore. If the semaphore is not

available and **NOWAIT** was not specified, then the task must be blocked until the semaphore is released. A proxy is allocated on the remote node to represent the task until the semaphore is released.

5. **SM_V – Release a semaphore**

CALLING SEQUENCE:

dir_status sm_v (smid)

INPUT:

obj_id smid; /* semaphore id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	semaphore released successfully
E_ID	invalid semaphore id

DESCRIPTION:

This directive releases the semaphore specified by **smid**. The semaphore count is incremented by one. If the count is zero or negative, then the first task on this semaphore's wait queue is removed and unblocked. The unblocked task may preempt the running task if the running task's preemption mode is enabled and the unblocked task has a higher priority than the ring task.

NOTES:

The calling task may be preempted if it causes a higher priority task to be made ready for execution.

Releasing a global semaphore which does not reside on the local node will generate a request telling the remote node to release the semaphore.

If the task to be unblocked resides on a different node from the semaphore, then the semaphore allocation is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

VIII. MESSAGE MANAGER

A. Introduction

The **message manager** provides communication and synchronization capabilities using **RTEMS** message queues. The directives provided by the message manager are:

Name	Directive Description
q_create	**Create a queue**
q_ident	**Get ID of a queue**
q_delete	**Delete a queue**
q_send	**Put message at rear of a queue**
q_urgent	**Put message at front of a queue**
q_broadcast	**Broadcast N messages to a queue**
q_receive	**Receive message from a queue**
q_flush	**Flush all messages on a queue**

B. Background

1. Messages

A message is a fixed length buffer where information can be stored to support communication. A message has a length of sixteen bytes. The information stored in a message is user–defined and can be actual data, pointer(s), or empty.

2. Message Queues

A **message queue** permits the passing of messages among tasks and ISRs. Message queues can contain a variable number of messages. Normally messages are sent to and received from the queue in **FIFO** order using the **q_send** directive. However, the **q_urgent** directive can be used to place messages at the head of a queue in LIFO order.

Synchronization can be realized because a task can wait for a message to arrive at a queue. Also, a task may poll a queue for the arrival of a message.

3. Building an Attribute Set

In general, a attribute set is built by a bitwise OR of the desired attributes. The set of valid attributes is provided in the description of the **q_create** and **q_receive** directives. An attribute listed as a default is not required to appear in the attribute OR list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute **DEFAULTS** should be specified on this call.

This example demonstrates the **attr** parameter needed to create a local message queue with a task priority waiting queue discipline. The **attr** parameter could be **PRIORITY** or **LOCAL | PRIORITY**. The **attr** parameter can be set to **PRIORITY** because **LOCAL** is the default for all created message queues. If a similar message queue were to be known globally, then the **attr** parameter would be **GLOBAL | PRIORITY**.

C. Operations

1. Creating a Message Queue

The **q_create** directive creates a message queue with the user–defined name. Optionally, a limit can be placed on the number of messages allowed to be in the message queue at one time. The user may select FIFO or task priority as the method for placing waiting tasks in the task wait queue. **RTEMS** allocates a Queue Control Block (QCB) from the QCB free list to maintain the newly created queue. **RTEMS** also generates a message queue ID which is returned to the calling task.

2. Obtaining Message Queue IDs

When a message queue is created, **RTEMS** generates a unique message queue ID. The message queue ID may be obtained by either of two methods. First, as the result of an invocation of the **q_create** directive, the queue ID is stored in a user provided location. Second, the queue ID may be obtained later using the **q_ident** directive. The queue ID is used by other message manager directives to access this message queue.

3. Receiving a Message

The **q_receive** directive attempts to retrieve a message from the specified message queue. If at least one message is in the queue, then the message is removed from the queue, copied to the caller's message buffer, and returned immediately. When messages are unavailable, one of the following situations applies:

- *By default, the calling task will wait forever for the message to arrive.*
- *Specifying the* **NOWAIT** *option forces an immediate return with an error status code.*
- *Specifying a timeout limits the period the task will wait before returning with an error status.*

If the task waits for a message, then it is placed in the message queue's task wait queue in either FIFO or task priority order. All tasks waiting on a message queue are returned an error code when the message queue is deleted.

4. Sending a Message

Messages can be sent to a queue with the **q_send** and **q_urgent** directives. These directives work identically when tasks are waiting to receive a message. A task is removed from the task waiting queue, unblocked, and the message is copied to a waiting task's message buffer.

When no tasks are waiting at the queue, **q_send** places the message at the rear of the message queue, while **q_urgent** places the message at the front of the queue. The message is copied to a **RTEMS** message buffer and then placed in the message queue. Neither directive can successfully send a message to a full queue.

5. Broadcasting a Message

The **q_broadcast** directive sends the same message to every task waiting on the specified message queue as an atomic operation. The message is copied to each waiting

task's message buffer and each task is unblocked. The number of tasks which were unblocked is returned to the caller.

6. Message Queue Deletion

The **q_delete** directive removes a message queue from the system and frees its control block. A message queue can be deleted by any task that knows the message queue's ID. As a result of this directive, all tasks blocked waiting to receive a message from the message queue will be readied and returned a status code which indicates that the message queue was deleted. Any subsequent references to the message queue's name and ID are invalid. Any messages waiting at the message queue are also deleted and deallocated.

D. Directives

This section details the message manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **Q_CREATE – Create a queue**

CALLING SEQUENCE:

dir_status q_create (name, count, attr, & qid)

INPUT:

```
obj_name      name;    /* user–defined name     */
unsigned32    count;   /* max message count     */
unsigned32    attr; /  * queue attributes       */
obj_id        *qid;    /* queue id              */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	queue created successfully
E_TOOMANY	too many queues created
E_UNSATISFIED	out of message buffers
E_NOMP	multiprocessing not configured
E_TOOMANY	too many global objects

DESCRIPTION:

This directive creates a message queue which resides on the local node with the user–defined name specified in **name**. For control and maintenance of the queue, **RTEMS** allocates and initializes a QCB. The **RTEMS**–assigned queue id, returned in **qid**, is used to access the message queue.

Specifying **PRIORITY** in **attr** causes tasks waiting for a message to be serviced according to task priority. When **FIFO** is specified, waiting tasks are serviced in First In–First Out order.

If **LIMIT** is specified in **attr**, then a limit is fixed on the maximum number of message buffers that can be contained in the queue. Buffers are dynamically allocated from the system message buffer pool as needed. The **count** parameter is disregarded if **NOLIMIT** is specified.

NOTES:

This directive will not cause the calling task to be preempted.

If the **LIMIT** option is selected and **count** has a value of zero, then the **q_send** and **q–urgent** directives will fail unless one or more tasks are already waiting on the queue.

The following message queue attribute constants are defined by **RTEMS:**

CONSTANT	DESCRIPTION	DEFAULT
FIFO	tasks wait by FIFO	*
PRIORITY	tasks wait by priority	
NOLIMIT	unlimited queue size	*
UNIT	limit queue size to count	
LOCAL	local message queue	*
GLOBAL	global message queue	

Message queues should not be made global unless remote tasks must interact with the created message queue. This is to avoid the system overhead incurred by the creation of a global message queue. When a global message queue is created, the message queue's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including message queues, is limited by the **num_gobjects** field in the configuration table.

2. Q_IDENT – Get ID of a queue

CALLING SEQUENCE:

dir_status q_ident (name, node, & qid)

INPUT:

```
obj_name      name;    /* user–defined name    */
unsigned32    node;    /* node(s) to search    */
```

OUTPUT:

```
obj_id        *qid;    /* queue id
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	queue identified successfully
E_NAME	queue name not found
E_NODE	invalid node id

DESCRIPTION:

This directive obtains the queue id associated with the queue name specified in **name**. If the queue name is not unique, then the queue id will match one of the queues with that name. However, this queue id is not guaranteed to correspond to the desired queue. The queue id is used with other message related directives to access the message queue.

NOTES:

This directive will not cause the running task to be preempted.

If **node** is **ALL_NODES**, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If **node** is a valid node number which does not represent the local node, then only the message queues exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

3. Q_DELETE – Delete a queue

CALLING SEQUENCE:

dir_status q_delete (qid)

INPUT:

obj_id qid; /* queue id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	queue deleted successfully
E_ID	invalid queue id
E_REMOTE	cannot delete remote queue

DESCRIPTION

This directive deletes the message queue specified by **qid**. Any tasks that are waiting on this queue are unblocked with an error code for a deleted queue. If no tasks are waiting, but the queue contains messages, then **RTEMS** returns these message buffers back to the system message buffer pool. The QCB for this queue is reclaimed by **RTEMS**.

NOTES:

The calling task will be preempted if its preemption mode is enabled and one or more local tasks with a higher priority than the calling task are waiting on the deleted queue. The calling task will **NOT** be preempted if the tasks that are waiting are remote tasks.

The calling task does not have to be the task that created the queue, although the task and queue must reside on the same node.

When the queue is deleted, any messages in the queue are returned to the free message buffer pool. Any information stored in those messages is lost.

When a global message queue is deleted, the message queue id must be transmitted to every node in the system for deletion from the local copy of the global object table.

Proxies, used to represent remote tasks, are reclaimed when the message queue is deleted.

4. **Q_SEND – Put message at rear of a queue**

CALLING SEQUENCE:

dir_status q_send (qid, buffer)

INPUT:

```
obj_id    qid;          /* gueue id                 */
long      (*buffer)[4]; /* message buffer pointer   */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	message sent successfully
E_ID	invalid queue id
E_UNSATISFIED	out of message buffers
E_TOOMANY	queue's limit has been reached

DESCRIPTION:

This directive sends the message contained in **buffer** to the queue specified by **qid**. If a task is waiting at the queue, then the message is copied to the waiting task's buffer and the task is unblocked. If no tasks are waiting at the queue, then the message is copied to a message buffer which is obtained from **RTEMS'** message buffer pool. The message buffer is then placed at the rear of the queue.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending a message to a global message queue which does not reside on the local node will generate a request to the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

5. Q_URGENT – Put message at front of a queue

CALLING SEQUENCE:

dir_status q_urgent (qid, buffer)

INPUT:

```
obj_id    qid;          /* queue id                */
long      (*buffer)[4]; /* message buffer pointer  */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	message sent successfully
E_ID	invalid queue id
E_UNSATISFIED	out of message buffers
E_TOOMANY	queue's limit has been reached

DESCRIPTION:

This directive sends the message contained in **buffer** to the queue specified by **qid**. If a task is waiting on the queue, then the message is copied to the task's buffer and the task is unblocked. If no tasks are waiting on the queue, then the message is copied to a message buffer which is obtained from **RTEMS'** message buffer pool. The message buffer is then placed at the front of the queue.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending a message to a global message queue which does not reside on the local node will generate a request telling the remote node to post the message on the specified message queue.

If the task to be unblocked resides on a different node from the message queue, then the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

6. Q_BROADCAST – Broadcast N messages to a queue

CALLING SEQUENCE:

dir_status q_broadcast (qid, buffer, &count)

INPUT:

```
obj_id    qid;          /* queue id                */
long      (*buffer)[4]; /* message buffer pointer  */
```

OUTPUT:

```
unsigned32    *count;   /* tasks made ready        */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	message broadcasted successfully
E_ID	invalid queue id

DESCRIPTION:

This directive causes all tasks that are waiting at the queue specified by **qid** to be unblocked with the message contained in **buffer**. Before a task is unblocked, the message in **buffer** is copied to that task's message buffer. The number of tasks that were unblocked is returned in **count**.

NOTES:

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

The cost of this directive is directly related to the number of tasks waiting on the message queue, although it is more efficient than the equivalent number of invocations of **q_send**.

Broadcasting a message to a global message queue which does not reside on the local node will generate a request telling the remote node to post the message on the specified message queue.

When a task is unblocked which resides on a different node from the message queue, a copy of the message is forwarded to the appropriate node, the waiting task is unblocked, and the proxy used to represent the task is reclaimed.

7. Q_RECEIVE – Receive message from a queue

CALLING SEQUENCE:

dir_status q_receive (qid, buffer, options, timeout)

INPUT:

```
obj_id      qid;          /* queue id                 */
long        (*buffer)[4]; /* message buffer pointer   */
unsigned32  options;      /* receive options          */
interval    timeout;      /* wait interval            */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	message received successfully
E_ID	invalid queue id
E_UNSATISFIED	queue is empty
E_TIMEOUT	timed out waiting for message
E_DELETE	queue deleted while waiting

DESCRIPTION:

This directive receives a message from the message queue specified in **qid**. The **WAIT** and **NOWAIT** options of the **options** parameter allow the calling task to specify whether to wait for a message to become available or return immediately. For either option, if there is at least one message in the queue, then it is copied to **buffer** and this directive returns immediately with a successful return code.

If the calling task chooses to return immediately and the queue is empty, then a status code indicating this condition is returned. If the calling task chooses to wait at the message queue and the queue is empty, then the calling task is placed on the message wait queue and blocked. If the queue was created with the **PRIORITY** option specified, then the calling task is inserted into the wait queue according to its priority. But, if the queue was created with the **FIFO** option specified, then the calling task is placed at the rear of the wait queue.

A task choosing to wait at the queue can optionally specify a timeout value in the timeout parameter. The **timeout** parameter specifies the minimum interval to wait before the calling task desires to be unblocked. If it is set to **NOTIMEOUT**, then the calling task will wait forever.

NOTES:

The following message receive option constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
WAIT	wait for message	*
NOWAIT	do NOT wait for message	

Receiving a message from a global message queue which does not reside on the local node will generate a request to the remote node to obtain a message from the specified message queue. If no message is available and **WAIT** was specified, then the task must be blocked until a message is posted. A proxy is allocated on the remote node to represent the task until the message is posted.

8. Q_FLUSH – Flush all messages on a queue

CALLING SEQUENCE:

dir_status q_flush (qid, & count)

INPUT:

obj_id qid; /* queue id /*

OUTPUT:

unsigned32 *count; /* messages flushed */

DIRECTIVE STATUS CODES:

SUCCESSFUL	message received successfu y
E_ID	invalid queue id

DESCRIPTION:

This directive removes all pending messages from the specified queue **qid**. The number of messages removed is returned in **count**. If no messages are present on the queue, count is set to zero.

NOTES:

Flushing all messages on a global message queue which does not reside on the local node will generate a request to the remote node to actually flush the specified message queue.

IX. EVENT MANAGER

A. Introduction

The **event manager** provides a high performance method of intertask communication and synchronization. The directives provided by the event manager are:

Name	Directive Description
ev_send	Send event set to a task
ev_receive	Receive event condition

B. Background

1. Event Sets

An **event flag** is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty–two event flags are associated with each task. A collection of one or more event flags is referred to as a event set. The application developer should remember the following key characteristics of event operations when utilizing the event manager:

- *Events provide a simple synchronization facility.*
- *Events are aimed at tasks.*
- *Tasks can wait on more than one event simultaneously.*
- *Events are independent of one another.*
- *Events do not hold or transport data.*
- *Events are not queued. In other words, if an event is sent more than once before being received, the second and subsequent send operations have no effect.*

An event set is **posted** when it is directed (or sent) to a task. A **pending event** is an event that has been posted but not received. An event condition is used to specify the events which the task desires to receive id the algorithm which will be used to determine when the request is satisfied. An event condition is **satisfied** based upon one of two algorithms which are selected by the user. The **ANY** algorithm states that an event condition is satisfied when at least a single requested event is posted. The **ALL** algorithm states that an event condition is satisfied when every requested event is posted.

2. Building an Event Set or Condition

An event set or condition is built by a bitwise OR of the desired events. The set of valid events is **EVENT_0** through **EVENT_31**. If an event is not explicitly specified in the set or condition, then it is not present.

For example, when sending the event set consisting of **EVENT_6**, **EVENT_15**, and **EVENT_31**, the **event** parameter to the **ev_send** directive should be **EVENT_6 | EVENT_15 | EVENT_31**.

3. Building a Flag

In general, a flag is built by a bitwise OR of the desired options. The set of valid options is provided in the description of the **ev_receive** directive. An option listed as a default is not required to appear in the option OR list, although it is a good program practice to specify default options. If all defaults are desired, the option **DEFAULTS** should be specified on this call.

This example demonstrates the **flag** parameter needed to poll for all events in a particular event condition to arrive. The **flag** parameter should be **ALL | NOWAIT** or **NOWAIT**. The **flag** parameter can be set to **NOWAIT** because **ALL** is the default condition for **ev_receive**.

C. Operations

1. Sending an Event Set

The **ev_send** directive allows a task (or an ISR) to direct an event set to a target task. Based upon the state of the target task, one of the following situations applies:

- *Target Task is Blocked Waiting for Events*
  - If the waiting task's input event condition is satisfied, then the task is made ready for execution.
  - If the waiting task's input event condition is not satisfied, then the event set is posted but left pending and the task remains blocked.
- *Target Task is Not Waiting for Events*
  - The event set is posted and left pending.

2. Receiving an Event Set

The **ev_receive** directive is used by tasks to accept a specific input event condition. The task also specifies whether the request is satisfied when all requested events are available or any single requested event is available. If the requested event condition is satisfied by pending events, then a successful return code and the satisfying event set are returned immediately. If the condition is not satisfied, then one of the following situations applies:

- *By default, the calling task will wait forever for the event condition to be satisfied.*
- *Specifying the* **NOWAIT** *option forces an immediate return with an errors status code.*
- *Specifying a timeout limits the period the task will wait before returning with an error status code.*

3. Determining the Pending Event Set

A task can determine the pending event set by calling the **ev_receive** directive with a value of **CURRENT** for the input event condition. The pending events are returned to the calling task but the event set is left unaltered.

D. Directives

This section details the event manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **EV_SEND – Send event set to a task**

CALLING SEQUENCE:

dir_status ev_send (tid, event)

INPUT:

```
obj_id      tid;     /* task id              */
event_set   event;   /* event set to send    */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	event set sent successfully
E_ID	invalid task id

DESCRIPTION:

This directive sends an event set, **event**, to the task specified by **tid**. If a blocked task's input event condition is satisfied by this directive, then it will be made ready. If its input event condition is not satisfied, then the events satisfied are updated and the events not satisfied are left pending. If the task specified by tid is not blocked waiting for events, then the events sent are left pending.

NOTES:

Specifying **SELF** for **tid** results in the event set being sent to the calling task.

Identical events sent to a task are not queued. In other words, the second, and subsequent, posting of an event to a task before it can perform an **ev_receive** has no effect.

The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive.

Sending an event set to a global task which does not reside on the local node will generate a request telling the remote node to send the event set to the appropriate task.

2. EV_RECEIVE – Receive event condition

CALLING SEQUENCE:

dir_status ev_receive (eventin, options, timeout, & eventout)

INPUT:

```
event_set     eventin;  /* input event condition  */
unsigned32    options;  /* receive options        */
interval      timeout;  /* wait interval          */
```

OUTPUT:

```
event_set *eventout;    /* output event set       */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL event received successfully
E_UNSATISFIED input event not satisfied (NOWAIT)
E_TIMEOUT timed out waiting for event

DESCRIPTION:

This directive attempts to receive the event condition specified in **eventin**. If **eventin** is set to **CURRENT**, then the current pending events are returned in **eventout** and left pending. The **WAIT** and **NOWAIT** options in the options parameter are used to specify whether or not the task is willing to wait for the event condition to be satisfied. **EV_ANY** and **EV_ALL** are used in the **options** parameter are used to specify whether a single event or the complete event set is necessary to satisfy the event condition. The **eventout** parameter is returned to the calling task with the value that corresponds to the events in **eventin** that were satisfied.

If pending events satisfy the event condition, then **eventout** is set to the satisfied events and the pending events in the event condition are cleared. If the event condition is not satisfied and **NOWAIT** is specified, then **eventout** is set to the currently satisfied events. If the calling task chooses to wait, then it will block waiting for the event condition.

If the calling task must wait for the event condition to be satisfied, then the **timeout** parameter is used to specify the maximum interval to wait. If it is set to **NOTIMEOUT**, then the calling task will wait forever.

NOTES:

This directive only affects the events specified in **eventin**. Any pending events that do not correspond to any of the events specified in **eventin** will be left pending.

The following event receive option constants are defined by **RTEMS**;

CONSTANT	DESCRIPTION	DEFAULT
WAIT	task will wait for event	*
NOWAIT	task should not wait	
EV_ALL	return after all events	*
EV_ANY	return after any events	

X. SIGNAL MANAGER

A. Introduction

The **signal manager** provides the capabilities required for asynchronous communication. The directives provided by the signal manager are:

Name	Directive Description
as_catch	Establish an ASR
as_send	Send signal set to a task

B. Background

1. Definitions

The signal manager allows a task to optionally define an **Asynchronous Signal Routine** (ASR). An ASR is to a task what an ISR is to an application's set of tasks. When the processor is interrupted, the execution of an application is asso interrupted and an ISR is given control. Similarly, when a signal is sent to a task, that task's execution path will be "interrupted" by the ASR. Sending a signal to a task has no effect on the receiving task's current execution state.

A **signal flag** is used by a task (or ISR) to inform another task of the occurrence of a significant situation. Thirty–two signal flags are associated with each task. A collection of one or more signals is referred to as a **signal set**. A signal set is **posted** when it is directed (or sent) to a task. A **pending signal** is a signal that has been sent to a task with a valid ASR, but has not been processed by that task's ASR.

2. A Comparison of ASRs and ISRs

The format of an ASR is similar to that of an ISR with the following exceptions:

- *ISRs are scheduled by the processor hardware. ASRs are scheduled by* **RTEMS**
- *ISRs do not execute in the context of a task and may invoke only a subset of directives. ASRs execute in the context of a task and may execute any directive.*
- *When an ISR is invoked, it is passed the vector number as its argument. When an ASR is invoked, it is passed the set as its argument.*
- *An ASR has a task mode which can be different from that of the task. An ISR does not execute as a task and, as a result, does not have a task mode.*

3. Building a Signal Set

A signal set is built by a bitwise OR of the desired signals. The set of valid signals is **SIGNAL_0** through **SIGNAL_31**. If a signal is not explicitly specified in the set or condition, then it is not present.

This example demonstrates the signal parameter used when sending the signal set consisting or **SIGNAL_6, SIGNAL_15,** and **SIGNAL_31.** The **signal** parameter provided to the **as_send** directive should be **SIGNAL_ 6 | SIGNAL_15 | SIGNAL_31.**

4. Building an ASR's Mode

In general, an ASR's mode is built by a bitwise OR of the desired options. The set of valid mode options is the same as those allowed with the **t_create** and **t_mode** directives. A complete list of mode options is provided in the description of the **as_catch** directive. An option listed as a default is not required to appear in the option OR list, although it is a good programming practice to specify default options. If all defaults are desired, the option **DEFAULTS** should be specified on this call.

This example demonstrates the **mode** parameter used with the **as_catch** to establish an ASR which executes at interrupt level three and is nonpreemptible. The **mode** should be set to **INTR(3) | NOPREEMPT** to indicate the desired processor mode and interrupt level.

C. Operations

1. Establishing an ASR

The **as_catch** directive establishes an ASR for the calling task. The address of the ASR and its execution mode are specified to this directive. The ASR's mode is distinct from the task's mode. For example, the task may allow preemption, while that task's ASR may have preemption disabled. Until a task calls **as_catch** the first time, its ASR is invalid, and no signal sets can be sent to the task.

A task may invalidate its ASR and discard all pending signals by calling as_catch with a value of **NULL_ASR** for the ASR's address. When a task's ASR is invalid, new signal sets sent to this task are discarded.

A task may disable ASR processing (NOASR) via the **t_mode** directive. When a task's ASR is disabled, the signals sent to it are left pending to be processed later when the ASR is enabled.

Any directive that can be called from a task can also be called from an ASR. A task is only allowed one active ASR. Thus, each call to **as_catch** replaces the previous one.

Normally, signal processing is disabled for the ASR's execution mode, but if signal processing is enabled for the ASR the ASR must be reentrant.

2. Sending a Signal Set

The **as_send** directive allows both tasks and ISRs to send signals to a target task. The target task and a set of signals are specified to the **as_send** directive. The sending of a signal to a task has no effect on the execution state of that task. If the task is not the currently running task, then the signals are left pending and processed by the task's ASR the next time the task is dispatched to run. The ASR is executed immediately before the task is dispatched. If the currently running task sends a signal to itself or is sent a signal from an ISR, its ASR is immediately dispatched to run provided signal processing is enabled.

If an ASR with signals enabled is preempted by another task or an ISR and a new signal set is sent, then a new copy of the ASR will be invoked, nesting the preempted ASR.

Upon completion of processing the new signal set, control will return to the preempted ASR. In this situation, the ASR must be reentrant.

Like events, identical signals sent to a task are not queued. In other words, sending the same signal multiple times to a task (without any intermediate signal processing occurring for the task), has the same result as sending that signal to that task once.

3. Processing an ASR

Asynchronous signals were designed to provide the capability to generate software interrupts. The processing of software interrupts parallels that of hardware interrupts. As a result, the differences between the formats of ASRs and ISRs is limited to the meaning of the single argument passed to an ASR. The ASR should have the following C calling sequence and adhere to C calling conventions:

```
void asr (signals)
unsigned32 signals;      /* signal set              /*
```

When the ASR returns to RTEMS the mode and execution path of the interrupted task (or ASR) is restored to the context prior to entering the ASR.

D. Directives

This section details the signal manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **AS_CATCH – Establish an ASR**

CALLING SEQUENCE:

dir_status as_catch (asraddr, mode)

INPUT:

```
asr_ptr      asraddr;  /* ASR entry point */
unsigned32   mode;     /* mode for ASR    */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive establishes an asynchronous signal routine (ASR) for the calling task. The **asraddr** parameter specifies the entry point of the ASR. If **asraddr** is **NULL_ASR**, the ASR for the calling task is invalidated and all pending signals are cleared. Any signals sent to a task with an invalid ASR are discarded. The **mode** parameter specifies the execution mode for the ASR. This execution mode supersedes the task's execution mode while the ASR is executing.

NOTES:

This directive will not cause the calling task to be preempted.

The following task mode constants are defined by RTEMS:

CONSTANT	DESCRIPTION	DEFAULT
PREEMPT	enable preemption	*
NOPREEMPT	disable preemption	
NOTSLICE	disable timeslicing	*
TSLICE	enable timeslicing	
ASR	enable ASR processing	*
NOASR	disable ASR processing	
INTR(0)	enable all interrupts	*
INTR(n)	execute at interrupt level n	

2. **AS_SEND – Send signal set to a task**

CALLING SEQUENCE:

dir_status as_send (tid, signal)

INPUT:

```
obj_id      tid;     /* task id              /*
signal_set  signal;  /* signal set to send   */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	signal sent successfully
E_ID	task id invalid
E_NOTDEFINED	ASR invalid

DESCRIPTION:

This directive sends a signal set to the task specified in **tid**. The **signal** parameter contains the signal set to be sent to the task.

If a caller sends a signal set to a task with an invalid ASR then an error code is returned to the caller. If a caller sends a signal set to a task whose ASR is valid but disabled, then the signal set will be caught and left pending for the ASR to process when it is enabled. If a caller sends a signal set to a task with an ASR that is both valid and enabled, then the signal set is caught and the ASR will execute the next time the task is dispatched to run.

NOTES:

Sending a signal set to a task has no effect on that task's state. If a signal set is sent to a blocked task, then the task will remain blocked and the signals will be processed when the task becomes the running task.

Sending a signal set to a global task which does not reside on the local node will generate a request telling the remote node to send the signal set to the specified task.

XI. PARTITION MANAGER

A. Introduction

The **partition manager** provides facilities to dynamically allocate memory in fixed–size units. The directives provided by the partition manager are:

Name	Directive Description
pt_create	Create a partition
pt_ident	Get ID of a partition
pt_delete	Delete a partition
pt_getbuf	Get buffer from a partition
pt_retbuf	Return buffer to a partition

B. Background

1. Definitions

A **partition** is a physically contiguous memory area divided into fixed–size buffers that can be dynamically allocated and deallocated.

Partitions are managed and maintained as a list of buffers. Buffers are obtained from the front of the partition's free buffer chain and returned to the rear of the same chain. When a buffer is on the free buffer chain, **RTEMS** uses eight bytes of each buffer as the free buffer chain. When a buffer is allocated, the entire buffer is available for application use. Therefore, modifying memory that is outside of an allocated buffer could destroy the free buffer chain or the contents of an adjacent allocated buffer.

2. Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attributes. The set of valid attributes is provided in the description of the **pt_create** directive. An attribute listed as a default is not required to appear in the attribute **OR** list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute **DEFAULTS** should be specified on this call. The **attr** parameter should be **GLOBAL** to indicate that the partition is to be known globally.

C. Operations

1. Creating a Partition

The **pt_create** directive creates a partition with a user–specified name. The partition's name, starting address, length and buffer size are all specified to the **pt_create** directive. **RTEMS** allocates a **Partition Control Block** (PTCB) from the PTCB free list. This data structure is used by **RTEMS** to manage the newly created partition. The number of buffers in the partition is calculated based upon the specified partition length and buffer size, and returned to the calling task along with a unique partition ID.

2. Obtaining Partition IDs

When a partition is created, **RTEMS** generates a unique partition ID and assigned it to the created partition until it is deleted. The partition ID may be obtained by either of two methods. First, as the result of an invocation of the **pt_create** directive, the partition ID is stored in a user provided location. Second, the partition ID may be obtained later using the **pt_ident** directive. The partition ID is used by other partition manager directives to access this partition.

3. Acquiring a Buffer

A buffer can be obtained by calling the **pt_getbuf** directive. If a buffer is available, then it is returned immediately with a successful return code.

Otherwise, an unsuccessful return code is returned immediately to the caller. Tasks cannot block to wait for a buffer to become available.

4. Releasing a Buffer

Buffers are returned to a partition's free buffer chain with the **pt_retbuf** directive. This directive returns an error status code if the returned buffer was not previously allocated from this partition.

5. Deleting a Partition

The **pt_delete** directive allows a partition to be removed and returned to RTEMS. When a partition is deleted, the PTCB for that partition is returned to the PTCB free list. A partition with buffers still allocated cannot be deleted. Any task attempting to do so will be returned an error status code.

D. Directives

This section details the partition manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **PT_CREATE – Create a partition**

CALLING SEQUENCE:

dir_status pt_create (name, paddr, length, bsize, attr, & ptid)

INPUT:

```
obj_name     name;     /*  user–defined name          */
unsigned8    *paddr;   /*  physical start address     */
unsigned32   length;   /*  physical length in bytes   */
unsigned32   bsize;    /*  buffer size in bytes       */
unsigned32   attr; /*  /*  partition attributes       */
```

OUTPUT:

```
obj_id       *ptid;    /* id assigned to partition    */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	partition created successfully
E_TOOMANY	too many partitions created
E_ADDRESS	address not on long–word boundary
E_SIZE	buffer size not a multiple of 4
E_N0MP	multiprocessing not configured
E_TOOMANY	too many global objects

DESCRIPTION:

This directive creates a partition of fixed size buffers from a physically contiguous memory space. The assigned partition id is returned in **ptid**. This partition id is used to access the partition with other partition related directives. For control and maintenance of the partition, **RTEMS** allocates a PTCB from the local PTCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

The **paddr** and **bsize** parameters must be multiples of four (long–word aligned).

Memory from the partition is not used by **RTEMS** to store the Partition Control Block.

The following partition attribute constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
LOCAL	local partition	*
GLOBAL	global partition	

The PTCB for a global partition is allocated on the local node. The memory space used for the partition must reside in shared memory.

Partitions should not be made global unless remote tasks must interact with the partition. This is to avoid the overhead incurred by the creation of a global partition. When a global partition is created, the partition's name and id must be transmitted to every node in the system for insertion in the local copy of the global object table.

The total number of global objects, including partitions, is limited by the **num_gobjects** field in the **Configuration Table**.

2. PT_IDENT – Get ID of a partition

CALLING SEQUENCE:

dir_status pt_ident (name, node, & ptid)

INPUT:

```
obj_name      name;    /* user–defined name    */
unsigned32    node;    /* node(s) to search    */
```

OUTPUT:

```
obj_id    *ptid;    /* partition id        */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	partition identified successfully
E_NAME	partition name not found
E_NODE	invalid node id

DESCRIPTION:

This directive obtains the partition id associated with the partition **name**. If the partition name is not unique, then the partition id will match one of the partitions with that name. However, this partition id is not guaranteed to correspond to the desired partition. The partition id is used with other partition related directives to access the partition.

NOTES:

This directive will not cause the running task to be preempted.

If **node** is **ALL_NODES**, all nodes are searched with the local node being searched first. All other nodes are searched with the lowest numbered node searched first.

If **node** is a valid node number which does not represent the local node, then only the partitions exported by the designated node are searched.

This directive does not generate activity on remote nodes. It accesses only the local copy of the global object table.

3. PT_DELETE – Delete a partition

CALLING SEQUENCE:

dir_status pt_delete (ptid)

INPUT:

obj_id ptid; /* partition id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	partition deleted successfully
E_ID	invalid partition id
E_INUSE	buffers still in use
E_REMOTE	cannot delete remote partition

DESCRIPTION:

This directive deletes the partition specified by **ptid**. The partition cannot be deleted if any of its buffers are still allocated. The PTCB for the deleted partition is reclaimed by **RTEMS**.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the partition. Any local task that knows the partition id can delete the partition.

When a global partition is deleted, the partition id must be transmitted to every node in the system for deletion from the local copy of the global object table.

The partition must reside on the local node, even if the partition was created with the **GLOBAL** option.

4. **PT_GETBUF – Get buffer from a partition**

CALLING SEQUENCE:

dir_status pt_getbuf (ptid, &bufaddr)

INPUT:

obj_id ptid; /* partition id */

OUTPUT:

unsigned8 **bufaddr; /* buffer address */

DIRECTIVE STATUS CODES:

SUCCESSFUL	buffer obtained successfully
E_ID	invalid partition id
E_UNSATISFIED	all buffers are allocated

DESCRIPTION:

This directive allows a buffer to be obtained from the partition specified in **ptid.**

The address of the allocated buffer is returned in **bufaddr.**

NOTES:

This directive will not cause the running task to be preempted. All buffers begin on a long–word boundary.

A task cannot wait on a buffer to become available.

Getting a buffer from a global partition which does not reside on the local node will generate a request telling the remote node to allocate a buffer from the specified partition.

5. PT_RETBUF – Return buffer to a partition

CALLING SEQUENCE:

dir_status pt_retbuf (ptid, bufaddr)

INPUT:

```
obj_id      ptid;       /* partition id     /*
unsigned8   *bufaddr;   /* buffer to return */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	buffer returned successfully
E_ID	invalid partition id
E_ADDRESS	buffer address not in partition

DESCRIPTION:

This directive returns the buffer specified by **bufaddr** to the partition specified by **ptid.**

NOTES:

This directive will not cause the running task to be preempted.

Returning a buffer to a global partition which does not reside on the local node will generate a request telling the remote node to return the buffer to the specified partition.

XII. REGION MANAGER

A. Introduction

The **region manager** provides facilities to dynamically locate memory in variable sized units. The directives provided by the region manager are:

Name	Directive Description
rn_create	Create a region
rn_ident	Get ID of a region
rn_delete	Delete a region
rn_getseg	Get segment from a region
rn_retseg	Return segment to a region

B. Background

1. Definitions

A **region** makes up a physically contiguous memory space with user–defined boundaries from which variable–sized segments are dynamically allocated and deallocated. A **segment** is a variable size section of memory which is allocated in multiples of a user–defined page size. This page size is required to be a multiple of four better than or equal to four. For example, if a request for a 350–byte segment is made in a region with 256–byte pages, then a 512–byte segment is allocated.

Regions are organized as doubly linked chains of variable sized memory blocks. Memory requests are allocated using a first–fit algorithm. If available, the requester receives the number of bytes requested (rounded up to the next page size). **RTEMS** requires some overhead from the region's memory for each segment that is allocated. Therefore, an application should only modify the memory of a segment that has been obtained from the region. The application should **NOT** modify the memory outside of any obtained segments and within the region's boundaries while the region is currently active in the system.

Upon return to the heap, the free block is coalesced with its neighbors (if free) on both sides to produce the largest possible unused block.

2. Building an Attribute Set

In general, an attribute set is built by a bitwise OR of the desired attributes. The set of valid attributes is provided in the description of the **rn_create** and **rn_getseg** directives. An attribute listed as a default is not required to appear in the attribute OR list, although it is a good programming practice to specify default attributes. If all defaults are desired, the attribute **DEFAULTS** should be specified on this call.

For example, the **attr** parameter should be **PRIORITY** to indicate that task priority should be used as the task waiting queue discipline.

C. Operations

1. Creating a Region

The **rn_create** directive creates a region with the user–defined name. The user may select **FIFO** or task priority as the method for placing waiting tasks in the task wait queue. **RTEMS** allocates a **Region Control Block** (RNCB) from the RNCB free list to maintain the newly created region. **RTEMS** also generates a unique region ID which is returned to the calling task.

It is not possible to calculate the exact number of bytes available to the user since **RTEMS** requires overhead for each segment allocated. For example, a region with one segment that is the size of the entire region has more available bytes than a region with two segments that collectively are the size of the entire region. The reason is that the region with one segment requires only the overhead for one segment, while the other region requires the overhead for two segments.

Due to automatic coalescing, the number of segments in the region dynamically changes. Therefore, the total overhead required by **RTEMS** dynamically changes.

2. Obtaining Region IDs

When a region is created, **RTEMS** generates a unique region **ID** and assigns it to the created region until it is deleted. The region **ID** may be obtained by either of two methods. First, as the result of an invocation of the **rn_create** directive, the region **ID** is stored in a user provided location. Second, the region **ID** may be obtained later using the **rn_ident** directive. The region ID is used by other region manager directives to access this region.

3. Acquiring a Segment

The **rn_getseg** directive attempts to acquire a segment from a specified region. If the region has enough available free memory, then a segment is returned successfully to the caller. When the segment cannot be allocated, one of the following situations applies:

- *By default, the calling task will wait forever to acquire the segment.*
- *Specifying the* **NOWAIT** *option forces an immediate return with an error status code.*
- *Specifying a timeout limits the interval the task will wait before returning with an error status code.*

If the task waits for the segment, then it is placed in the region's task wait queue in either FIFO or task priority order. All tasks waiting on a region are returned an error when the message queue is deleted.

4. Releasing a Segment

When a segment is returned to a region by the **rn_retseg** directive, it is merged with its unallocated neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

5. Deleting a Region

A region can be removed from the system and returned to RTEMS with the **rn_delete** directive. When a region is deleted, its control block is returned to the RNCB free list. A region with segments still allocated is not allowed to be deleted. Any task attempting to do so will be returned an error.

D. Directives

This section details the region manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **RN_CREATE – Create a region**

CALLING SEQUENCE:

dir_status

rn_create (name, paddr, length, pagesize, attr,& rnid)

INPUT:

```
obj_name     name;     /* user–defined name     */
unsigned8    *paddr;   /* start address         */
unsigned32   length;   /* length in bytes       */
unsigned32   pagesize; /* page size in bytes    */
unsigned32   attr;     /* region attributes     */
```

OUTPUT:

```
obj_id       *rnid;    /* region id             */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	region created successfully
E_ADDRESS	address not on long–word boundary
E_TOOMANY	too many regions created
E_SIZE	invalid page size

DESCRIPTION:

This directive creates a region from a physically contiguous memory space. The assigned region id is returned in **rnid**. This region id is used as an argument to other region related directives to access the region.

For control and maintenance of the region, **RTEMS** allocates and initializes an RNCB from the RNCB free pool. Thus memory from the region is not used to store the RNCB. However, some overhead within the region is required by **RTEMS** each time a segment is constructed in the region.

Specifying **PRIORITY** in **attr** causes tasks waiting for a segment to be serviced according to task priority. Specifying **FIFO** in **attr** or selecting **DEFAULTS** will cause waiting tasks to be serviced in First In–First Out order.

The **paddr** parameter must be long–word aligned. The **pagesize** parameter must be a multiple of four greater than or equal to four.

NOTES:

This directive will not cause the calling task to be preempted. The following region attribute constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
FIFO	tasks wait by FIFO	*
PRIORITY	tasks wait by priority	

2. RN_IDENT – Get ID of a region

CALLING SEQUENCE:

dir_status rn_ident (name, & rnid)

INPUT:

```
obj_name name;     /* user-defined name     */
```

OUTPUT:

```
obj_id    *rnid;   /* region id             */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	region identified successfully
E_NAME	region name not found

DESCRIPTION:

This directive obtains the region id associated with the region **name** to be acquired. If the region name is not unique, then the region id will match one of the regions with that name. However, this region id is not guaranteed to correspond to the desired region. The region id is used to access this region in other region related directives.

NOTES:

This directive will not cause the running task to be preempted.

3. **RN_DELETE – Delete a region**

CALLING SEQUENCE:

dir_status rn_delete (rnid)

INPUT:

obj_id rnid; /* region id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	region deleted successfully
E_ID	invalid region id
E_INUSE	segments still in use

DESCRIPTION:

This directive deletes the region specified by **rnid**. The region cannot be deleted if any of its segments are still allocated. The RNCB for the deleted region is reclaimed by **RTEMS.**

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the region. Any local task that knows the region id can delete the region.

4. **RN_GETSEG – Get segment from a region**

CALLING SEQUENCE:

dir_status rn_getseg (rnid, size, options, timeout, &segaddr)

INPUT:

```
obj_id      rnid;     /* region id              */
unsigned32  size;     /* segment size in bytes  */
unsigned32  options;  /* option set             */
interval    timeout;  /* wait interval          */
```

OUTPUT:

```
unsigned8 **segaddr;  /* segment address        */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	segment obtained successfully
E_ID	invalid region id
E_SIZE	request exceeds size of maximum segment
E_UNSATISFIED	segment of requested size not available
E_TIMEOUT	timed out waiting for segment

DESCRIPTION:

This directive obtains a variable size segment from the region specified by **rnid**. The address of the allocated segment is returned in **segaddr**. The **WAIT** and **NOWAIT** options of the options parameter are used to specify whether the calling tasks wish to wait for a segment to become available or return immediately if no segment is available. For either option, if a sufficiently sized segment is available, then the segment is successfully acquired by returning immediately with the **SUCCESSFUL** status code.

If the calling task chooses to return immediately and a segment large enough is not available, then an error code indicating this fact is returned. If the calling task chooses to wait for the segment and a segment large enough is not available, then the calling task is placed on the region's segment wait queue and blocked. If the region was created with the **PRIORITY** option, then the calling task is inserted into the wait queue according to its priority. But, if the region was created with the **FIFO** option, then the calling task is placed at the rear of the wait queue.

The timeout parameter specifies the maximum interval that a task is wiling to wait to obtain a segment. If timeout is set to **NOTIMEOUT**, then the calling task will wait forever.

NOTES:

The actual length of the allocated segment may be larger than the requested size because a segment size is always a multiple of the region's pagesize.

The following segment acquisition option constants are defined by **RTEMS**:

CONSTANT	DESCRIPTION	DEFAULT
WAIT	task may wait for segment	*
NOWAIT	task may not wait	

5. RN_RETSEG – Return segment to a region

CALLING SEQUENCE:

dir_status rn_retseg (rnid, segaddr)

INPUT:

```
obj_id      rnid;       /* region id       */
unsigned8   *segaddr;   /* segment pointer */
```

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	segment returned successfully
E_ID	invalid region id
E_ADDRESS	segment address not in region

DESCRIPTION:

This directive returns the segment specified by **segaddr** to the region specified by **rnid**. The returned segment is merged with its neighbors to form the largest possible segment. The first task on the wait queue is examined to determine if its segment request can now be satisfied. If so, it is given a segment and unblocked. This process is repeated until the first task's segment request cannot be satisfied.

NOTES:

This directive will cause the calling task to be preempted if one or more local tasks are waiting for a segment and the following conditions exist:

- *a waiting has a higher priority than the calling task*
- *the size of the segment required by the waiting task is less than or equal to the size of the segment returned.*

XIII. DUAL–PORTED MEMORY MANAGER

A. Introduction

The **dual–ported memory manager** provides a mechanism for converting addresses between internal and external representations for multiple Dual–Ported Memory Areas (DPMA). The directives provided by the dual–ported memory manager are:

Name	Directive Description
dp_create	Create a port
dp_ident	Get ID of a port
dp_delete	Delete a port
dp_2internal	Convert external to internal address
dp_2external	Convert internal to external address

B. Background

A Dual–Ported Memory Area (DPMA) is an contiguous block of RAM owned by a particular processor but which can be accessed by other processors in the system. The owner accesses the memory using internal addresses, while other processors must use external addresses. **RTEMS** defines a **port** as a particular mapping of internal and external addresses.

There are two system configurations in which dual–ported memory is commonly found. The first is tightly–coupled multiprocessor computer systems where the dual–ported memory is shared between all nodes and is used for inter–node communication. The second configuration is computer systems with intelligent peripheral controllers. These controllers typically utilize the DPMA for high–performance data transfers.

C. Operations

1. Creating a Port

The **dp_create** directive creates a port into a DPMA with the user–defined name. The user specifies the association between internal and external representations for the port being created. **RTEMS** allocates a **Dual–Ported Memory Control Block** (DPCB) from the DPCB free list to maintain the newly created DPMA. **RTEMS** also generates a unique dual–ported memory port ID which is returned to the calling task. **RTEMS** does not initialize the dual–ported memory area or access any memory within it.

2. Obtaining Port IDs

When a port is created, **RTEMS** generates a unique port ID and assigns it to the created port until it is deleted. The port ID may be obtained by either of two methods. First, as the result of an invocation of the **dp_create** directive, the task ID is stored in a user provided location. Second, the port ID may be obtained later using the **dp_ident** directive. The port ID is used by other dual–ported memory manager directives to access this port.

3. Converting an Address

The **dp_2internal** directive is used to convert an address from external to internal representation for the specified port. The **dp_2external** directive is used to convert an address from internal to external representation for the specified port. If an attempt is made to convert an address which lies outside the specified DPMA, then the address to be converted will be returned.

4. Deleting a DPMA Port

A port can be removed from the system and returned to **RTEMS** with the **dp_delete** directive. When a port is deleted, its control block is returned to the DPCB free list.

D. Directives

This section details the dual–ported memory manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **DP_CREATE – Create a port**

CALLING SEQUENCE:

dir_status dp_create (name, intaddr, extaddr, length, &dpid)

INPUT:

```
obj_name      name;      /* user–defined name      */
unsigned8     *intaddr;  /* initial internal address */
unsigned8     *extaddr;  /* initial external address */
unsigned32    length;    /* area size in bytes     */
```

OUTPUT:

```
obj_id        *dpid;     /* port id                */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	port created successfully
E_ADDRESS	internal or external address not on long–word boundary
E_TOOMANY	too many DP memory areas created

DESCRIPTION:

This directive creates a port which resides on the local node for the specified DPMA. The assigned port id is returned in dpid. This port id is used as an argument to other dual–ported memory manager directives to convert addresses within this DPMA.

For control and maintenance of the port, RTEMS allocates and initializes an DPCB from the DPCB free pool. Thus memory from the dual–ported memory area is not used to store the DPCB.

NOTES:

The **intaddr** and **extaddr** parameters must be long–word aligned.

This directive will not cause the calling task to be preempted.

2. DP_IDENT – Get ID of a port

CALLING SEQUENCE:

dir_status dp_ident (name, &dpid)

INPUT:

```
obj_name        name;       /* user–defined name    */
```

OUTPUT:

```
obj_id          *dpid;      /* port id              */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	port identified successfully
E_NAME	port name not found

DESCRIPTION:

This directive obtains the port id associated with the specified **name** to be acquired. If the port name is not unique, then the port id will match one of the DPMAs with that name. However, this port id is not guaranteed to correspond to the desired DPMA. The port id is used to access this DPMA in other dual–ported memory area related directives.

NOTES:

This directive will not cause the running task to be preempted.

3. DP_DELETE – Delete a port

CALLING SEQUENCE:

dir_status dp_delete (dpid)

INPUT:

obj_id dpid; /* port id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	port deleted successfully
E_ID	invalid port id

DESCRIPTION:

This directive deletes the dual–ported memory area specified by dpid. The DPCB for the deleted dual–ported memory area is reclaimed by RTEMS.

NOTES:

This directive will not cause the calling task to be preempted.

The calling task does not have to be the task that created the port. Any local task that knows the port id can delete the port.

4. DP_2INTERNAL – Convert external to internal address

CALLING SEQUENCE:

dir_status dp_2internal (dpid, extaddr, &intaddr)

INPUT:

```
obj_id          dpid;       /* port id              */
unsigned8       *extaddr;   /* address to convert   */
```

OUTPUT:

```
unsigned8       **intaddr;  /* internal address     */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive converts a dual–ported memory address from external to internal representation for the specified **port**. If the given external address is invalid for the specified port, then the internal address is set to the given external address.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

5. **DP_2EXTERNAL – Convert internal to external address.**

CALLING SEQUENCE:

dir_status dp_2external (dpid, intaddr, &extaddr)

INPUT:

```
obj_id          dpid;       /* port id              */
unsigned8       *intaddr;   /* address to convert   */
```

OUTPUT:

```
unsigned8       **extaddr;  /* external address     */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL always successful

DESCRIPTION:

This directive converts a dual–ported memory address from internal to external representation so that it can be passed to owner of the DPMA represented by the specified **port.** If the given internal address is an invalid dual–ported address, then the external address is set to the given internal address.

NOTES:

This directive is callable from an ISR.

This directive will not cause the calling task to be preempted.

XIV. I/O MANAGER

A. Introduction

The **input/output interface manager** provides a well–defined mechanism for accessing device drivers and a structured methodology for organizing device drivers. The directives provided by the I/O manager are:

Name	Directive Description
de_init	Initialize a device driver
de_open	Open a device
de_close	Close a device
de_read	Read from a device
de_write	Write to a device
de_cntrl	Special device services

B. Background

1. Device Driver Table

Each application utilizing the **RTEMS** I/O manager must specify the address of a **Device Driver Table** in its **Configuration Table**. This table contains each device *driver's entry points*. Each device driver may contain the following entry points:

- *Initialization*
- *Open*
- *Close*
- *Read*
- *Write*
- *Control*

If the device driver does not support a particular entry point, then that entry in the **Configuration Table** should be **NULL_DRIVER**. **RTEMS** will return **SUCCESSFUL** as the executive's and device driver's return code for these device driver entry points.

2. Major and Minor Device Numbers

Each call to the I/O manager must provide a device number as an argument. This device number is a 32–bit unsigned entity composed of a major and a minor device number. The most significant sixteen bits are the major number, and the least significant sixteen bits compose the minor number. The major number is the index of the requested driver's entry points in the **Device Driver Table**, and is used to select a specific device driver. The exact usage of the minor number is driver specific, but is commonly used to distinguish between a number of devices controlled by the same driver.

3. Device Driver Environment

Application developers, as well as device driver developers, must be aware of the following regarding the **RTEMS** I/O Manager:

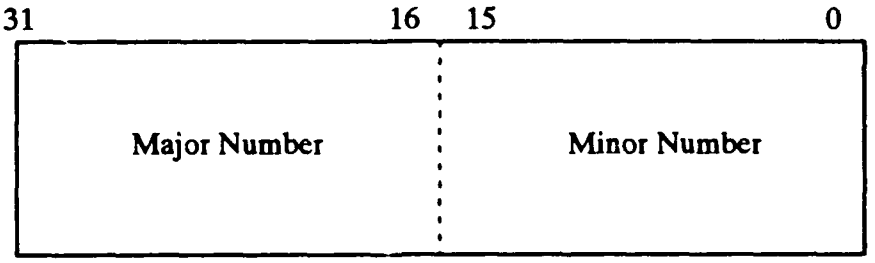

Figure 6. Device Number Composition

- *A device driver routine executes in the context of the invoking task. Thus, if the driver blocks, the invoking task blocks.*
- *The device driver is free to chant the modes of the invoking task, al though the driver should restore them to their original values.*
- *Device drivers can* **NOT** *be invoked from ISRs.*
- *Only local device drivers are accessible through the I/O manager.*
- *A device driver routine may invoke all other* **RTEMS** *directives, including I/O directives, on both local ana global objects.*

Although the **RTEMS** I/O manager provides a framework for device drivers, it makes no assumptions regarding the construction or operation of a device driver.

4. Device Driver Interface

When an application invokes I/O manager directive, **RTEMS** determines which device driver entry point must be invoked. The information passed by the application to **RTEMS** is then passed to the correct device driver entry point. **RTEMS** will invoke each device driver entry point with the following C calling sequence:

```
void de_entry( dev, argp, tid, rval )
unsigned32    dev;     /* device number              */
unsigned8     *argp;   /* parameter block address    */
obj_id        tid;     /* ID of invoking task        */
unsigned32    *rval;   /* driver's status area       */
```

The format and contents of the parameter block are device driver and entry point dependent.

It is recommended that a device driver avoid generating error codes which conflict with those used by **RTEMS**. A common technique used to generate driver specific error codes is to logically OR the driver's major number with an error code.

5. Device Driver Initialization

RTEMS automatically initializes all device drivers when multitasking is initiated via the **init_exec** directive. **RTEMS** initializes the device drivers by invoking each device driver initialization entry point with the following parameters:

dev corresponds to the major device number for this device driver with a minor device number of zero.

argp will point to the **Configuration Table**.

tid will contain zero.

The returned **rval** will be ignored by **RTEMS**. If the driver cannot successfully initialize the device, then it should invoke the fatal error manager.

C. Operations

The I/O manager provides directives which enable the application program to utilize device drivers in a standard manner. There is a direct correlation between the **RTEMS** I/O manager directives and the underlying device driver entry points.

D. Directives

This section details the I/O manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **DE_INIT – Initialize a device driver**

CALLING SEQUENCE:

dir_status de_init (dev, argp, &rval)

INPUT:

```
unsigned32    dev;      /* 32–bit device number      */
unsigned8     *argp;    /* address of a driver       */
                        /* specific parameter block  */
```

OUTPUT:

```
unsigned32    *rval;    /* return value from driver  */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	successfully initialized
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver initialization routine specified in the **Device Driver Table** for this major number. This directive is automatically invoked for each device driver when multitasking is initiated via the **init_exec** directive.

A device driver initialization module is responsible for initializing all hardware and data structures associated with a device. If necessary, it can allocate memory to be used during other operations.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being initialized.

2. **DE_OPEN – Open a device**

CALLING SEQUENCE:

dir_status de_open (dev, argp, & rval)

INPUT:

```
unsigned32      dev;        /* 32–bit device number */
unsigned8       *argp;      /* address of a driver  */
                            /* specific parameter block */
```

OUTPUT:

```
unsigned32      *rval;      /* return value from driver */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	device successfully opened
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver open routine specified in the **Device Driver Table** for this major number. The open entry point is commonly used by device drivers to provide exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

3. DE_CLOSE – Close a device

CALLING SEQUENCE:

dir_status de_close (dev, argp, & rval)

INPUT:

```
unsigned32      dev;        /* 32–bit device number      */
unsigned8       *argp;      /* address of a driver       */
                            /* specific parameter block */
```

OUTPUT:

```
unsigned32      *rval;      /* return value from drive   */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	device successfully closed
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver close routine specified in the **Device Driver Table** for this major number. The close entry point is commonly used by device drivers to relinquish exclusive access to a device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

4. **DE_READ – Read from a device**

CALLING SEQUENCE:

dir_status de_read (dev, argp, & rval)

INPUT:

```
unsigned32      dev;        /* 32–bit device number      */
unsigned8       *argp;      /* address of a driver       */
                            /* specific parameter block  */
```

OUTPUT:

```
unsigned32      *rval;      /* return value from driver  */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	device successfully read
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver read routine specified in the **Device Driver Table** for this major number. Read operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be replaced with data from the device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

5. **DE_WRITE – Write to a device**

CALLING SEQUENCE:

dir_status de_write (dev, argp, & rval)

INPUT:

```
unsigned32      dev;        /* 32-bit device number      */
unsigned8       *argp;      /* address of a driver       */
                            /* specific parameter block  */
```

OUTPUT:

```
unsigned32      *rval;      /* return value from driver  */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	device successfully written to
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

This directive calls the device driver write routine specified in the **Device Driver Table** for this major number. Write operations typically require a buffer address as part of the argument parameter block. The contents of this buffer will be sent to the device.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

6. DE_CNTRL – Special device services

CALLING SEQUENCE:

dir_status de_cntrl (dev, argp, & rval)

INPUT:

```
unsigned32      dev;        /* 32–bit device number */
unsigned8       *argp;      /* address of a driver  */
                            /* specific parameter block*/
```

OUTPUT:

```
unsigned32      *rval;      /* return value from driver */
```

DIRECTIVE STATUS CODES:

SUCCESSFUL	control function was successful
E_CALLED	called from within an ISR
E_NUMBER	invalid major device number

DESCRIPTION:

This directive calls the device driver I/O control routine specified in the **Device Driver Table** for this major number. The exact functionality of the driver entry called by this directive is driver dependent. It should not be assumed that the control entries of two device drivers are compatible. For example, an RS–232 driver I/O control operation may change the baud rate of a serial line, while an I/O control operation for a floppy disk driver may cause a seek operation.

NOTES:

This directive may or may not cause the calling task to be preempted. This is dependent on the device driver being invoked.

XV. FATAL ERROR MANAGER

A. Introduction

The **fatal error manager** processes all fatal or irrecoverable errors. The directive provided by the fatal error manager is:

Name	Directive Description
k_fatal	Invoke the fatal error handler

B. Background

The fatal error manager is called upon detection of: irrecoverable error condition by either **RTEMS** or the application software. Fatal errors can be detected from three sources:

- *the executive* (**RTEMS**)
- *user system code*
- *user application code*

RTEMS automatically invokes the fatal error manager upon detection of an error it considers to be fatal. Similarly, the user should invoke the fatal error manager upon detection of a fatal error.

A user–supplied fatal error handler can be specified in the **User Extension Table** to provide access to debuggers and monitors which may be present on the target hardware. If configured, the fatal error manager will invoke a user–supplied fatal error handler. If no user handler is configured or if the user handler returns control to the fatal error manager, then the **RTEMS** default fatal error handler is invoked. In general, the default handler will disable all maskable interrupts, place the error code in a known place (either on the stack or in a register), and halt the processor. The precise actions of the **RTEMS** fatal error handler are processor dependent and are discussed in the **Default Fatal Error Processing** chapter of the **C Applications Supplement** document for a specific target processor.

C. Operations

D. Announcing a Fatal Error

The **k_fatal** directive is invoked when a fatal error is detected. This directive is responsible for invoking an optional user–supplied fatal error handler and/or the **RTEMS** fatal error handler. All fatal error handlers are passed an error code to describe the error detected.

Occasionally, an application requires more sophisticated fatal error processing such as passing control to a debugger. For these cases, a user–supplied fatal error handler can be specified in the **RTEMS** configuration table. The **User Extension Table** parameter **fatal** contains the address of the fatal error handler to be executed when the **k_fatal** directive is called. If the parameter is set to **NULL_EXTENSION** or if the configured fatal error handler returns to the executive, then the default handler provided by **RTEMS** is executed. This default handler will halt execution on the processor where the error occurred.

E. Directives

This section details the fatal error manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. K_FATAL – Invoke the fatal error handler

CALLING SEQUENCE:

void k_fatal (errcode)

INPUT:

unsigned32 errcode; /* fatal error code */

OUTPUT: NONE

DIRECTIVE STATUS CODES: NONE

DESCRIPTION:

This directive processes fatal errors. If the FATAL error extension is defined in the configuration table, then the user–defined error extension is called. If configured and the provided FATAL extension returns, then the RTEMS default error handler is invoked. This directive can be invoked by RTEMS or by the user's application code including initialization tasks, other tasks, and ISRs.

NOTES:

This directive supports local operations only.

Unless the user–defined error extension takes special actions such as restarting the calling task, this directive **WILL NOT RETURN** to the caller.

The user–defined extension for this directive may wish to initiate a global shutdown.

XVI. SCHEDULING CONCEPTS

A. Introduction

The concept of scheduling in real–time systems dictates the ability to provide immediate response to specific external events, particularly the necessity of scheduling particular tasks to run within a specified time limit after the occurrence of an event. For example, software embedded in life–support systems used to monitor hospital patients must take instant action if a change in the patient's status is detected.

The component of **RTEMS** responsible for providing this capability is appropriately called the scheduler. The scheduler's sole purpose is to allocate the all important resource of processor time to the various tasks competing for attention. The **RTEMS** scheduler allocates the processor using a priority–based, preemptive algorithm augmented to provide round–robin characteristics within individual priority groups. The goal of this algorithm is to guarantee that the task which is executing on the processor at any point in time is the one with the highest priority among all tasks in the **ready** state.

There are two common methods of accomplishing the mechanics of this algorithm. Both ways involve a list or chain of tasks in the ready state. One method is to randomly place tasks in the ready chain forcing the scheduler to scan the entire chain to determine which task receives the processor. The other method is to schedule the task by placing it in the proper place on the ready chain based on the designated scheduling criteria at the time it enters the ready state. Thus, when the processor is free, the first task on the ready chain is allocated the processor. **RTEMS** schedules tasks using the second method to guarantee faster response times to external events.

B. Scheduling Mechanisms

RTEMS provides four mechanisms which allow the user to impact the task scheduling process:

- *user–selectable task priority level*
- *task preemption control*
- *task timeslicing control*
- *manual round–robin selection*

Each of these methods provides a powerful capability to customize sets of tasks to satisfy the unique and particular requirements encountered in custom real–time applications. Although each mechanism operates independently, there is a precedence relationship which governs the effects of scheduling modifications. The evaluation order for scheduling characteristics is always priority, preemption mode, and timeslicing. When reading the descriptions of timeslicing and manual round–robin it is important to keep in mind that preemption (if enabled) of a task by higher priority tasks will occur as required, overriding the other factors presented in the description.

1. Task Priority

The most significant of these mechanisms is the ability for the user to assign a priority level to each individual task when it is created and to alter a task's priority at run–time. **RTEMS** provides 255 priority levels. Level 255 is the lowest priority and Level 1 is the highest.

When a task is added to the ready chain, it is placed behind all other tasks of the same priority. This rule provides a round–robin within priority group scheduling characteristic. This means that in a group of equal priority tasks, tasks will execute in the order they become ready or FIFO order. Even though there are ways to manipulate and adjust task priorities, the most important rule to remember is:

The **RTEMS** *scheduler will always select the highest priority task that is ready to run when allocating the processor to a task.*

---

2. Preemption

Another way the user can alter the basic scheduling algorithm is by manipulating the preemption bit in the mode parameter of individual tasks. If preemption is disabled for a task, then the task will not relinquish control of the processor until it terminates, blocks, or re–enables preemption. Even tasks which become ready to run and possess higher priority levels will not be allowed to execute. Note that the preemption setting has no effect on the manner in which a task is scheduled. It only applies once a task has control of the processor.

3. Timeslicing

Timeslicing or round–robin scheduling is an additional method which can be used to alter the basic scheduling algorithm. Like preemption, timeslicing is specified on a task by task basis. If timeslicing is enabled for a task, **RTEMS** will limit the amount of time the task can execute before the processor is allocated to another task. Each tick of the real–time clock reduces the currently *running task's timeslice*. When the execution time equals the timeslice, **RTEMS** will dispatch another task of the same priority to execute. If there are no other tasks of the same priority ready to execute, then the current task is allocated an additional timeslice and continues to run. Remember that a higher priority task will preempt the task (unless preemption is disabled) as soon as it is ready to run, even if the task has not used up its entire timeslice.

4. Manual Round–Robin

The final mechanism for altering the **RTEMS** scheduling algorithm is called manual round–robin. Manual round–robin is invoked by using the **tm_wkafter** directive with a time interval of **YIELD**. This allows a task to give up the processor and be immediately returned to the ready chain at the end of its priority group. If no other tasks of the same priority are ready to run, then the task does not lose control of the processor.

5. Dispatching Tasks

The dispatcher is the **RTEMS** component responsible for allocating the processor to a ready task. In order to allocate the processor to one task, it must be deallocated or retrieved from the task currently using it. This involves a concept called a context switch. To perform a context switch, the dispatcher saves the context of the current task and restores the context of the task which has been allocated to the processor. Saving and restoring a task's context is the storing/loading of all the essential information about a task to enable it to continue execution without any effects of the interruption. For example, the contents of a task's register set must be the same when it is given the processor as they were when it was taken away. All of the information that must be saved or restored for a context switch is located either in the TCB or on the task's stacks.

Tasks that utilize a numeric coprocessor and are created with the FP attribute require additional operations during a context switch. These additional operations are necessary to save and restore the floating point context of FP tasks. To avoid unnecessary save and restore operations, the state of the numeric coprocessor is only saved when an FP task is dispatched and that task was not the last task to utilize the coprocessor.

C. Task State Transitions

Tasks in an **RTEMS** system must always be in one of the five allowable task states. These states are: **executing, ready, blocked, dormant**, and **non–existent.**

A task occupies the **non–existent** state before a **t_create** has been issued on its behalf. A task enters the **non–existent** state from any other state in the system when it is deleted with the **t_delete** directive. While a task occupies this state it does not have a TCB or a task ID assigned to it; therefore, no other tasks in the system may reference this task.

When a task is created via the **t_create** directive it enters the **dormant** state. This state is not entered through any other means. Although the task exists in the system, it cannot actively compete for system resources. It will remain in the dormant state until it is started via the **t_start** directive, at which time it enters the **ready** state. The task is now permitted to be scheduled for the processor and to compete for other system resources.

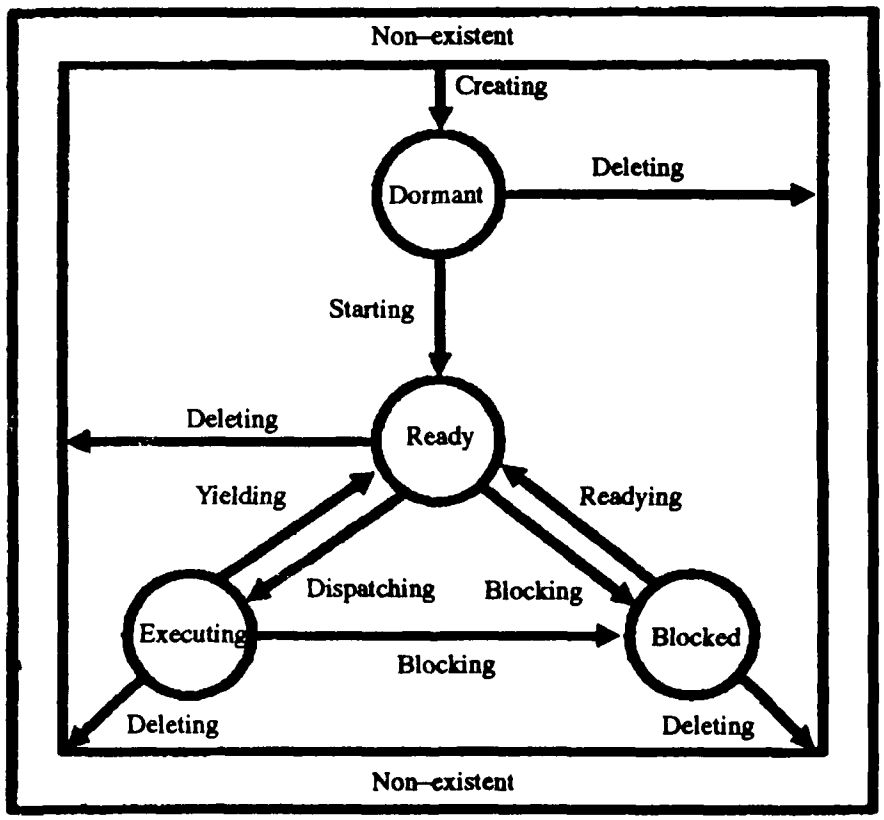

Figure 7. **RTEMS** *State Transitions*

A task occupies the **blocked** state whenever it is unable to be scheduled to run. A running task may block itself or be blocked by other tasks in the system. The ring task blocks itself through voluntary operations that cause the task to wait. The only way a task can block a task other than itself is with the **t_suspend** directive. A task enters the blocked state due to any of the following conditions:

- *A task issues a* **t_suspend** *directive which block either itself or another task in the system.*
- *The running task issues a* **q_receive** *directive with the wait option and the message queue is empty.*
- *The running task issues an* **ev_receive** *directive with the wait option and the currently pending events do not satisfy the request.*
- *The running task issues a* **sm_p** *directive with the wait option and the requested semaphore is unavailable.*
- *The running task issues a* **tm_wkafter** *directive which blocks the task for the even tune interval. If the time interval specified is zero, the task yields the processor and remains in the ready state.*
- *The running task issues a* **tm_wkwhen** *directive which blocks the task until the requested date and time arrives.*
- *The running task issues a* **rn_getseg** *directive with the wait option and there is not an available segment large enough to satin the task's request.*
- *The running task issues a* **rm_period** *directive and must wait for the specified rate monotonic timer to conclude.*

A blocked task may also be suspended. Therefore, both the suspension and the condition that caused the task to block, must be lifted before the task becomes ready to run.

A task occupies the **ready** state when it is able to be scheduled to run, but currently does not have control of the processor. Tasks of the same or higher priority will yield the processor by either becoming blocked, completing their timeslice, or being deleted. All tasks with the same priority will execute in **FIFO** order. A task enters the ready state due to any of the following conditions:

- *A running task issues a* **t_resume** *directive for a task that is suspended and the task is not blocked waiting on any resource.*
- *A running task issues a* **q_send, q_broadcast,** *or a* **q_urgent** *directive which posts a message to the queue on which the blocked task is waiting.*
- *A running task issues an* **ev_send** *directive which sends an event condition to a task which is blocked waiting on that event condition.*
- *A running task issues a* **sm_v** *directive which releases the semaphore on which the blocked task is waiting.*
- *A timeout interval expires for a task which was blocked by a call to the* **tm_wkafter** *directive.*
- *A timeout period expires for a task which blocked by a call to the* **tm_wkwhen** *directive.*

- *A running task issues a* **rn_retseg** *directive which releases a segment to the region on which the blocked task is waiting and a resulting segment is large enough to satisfy the task's request.*
- *A rate monotonic timer expires for a task which blocked by a call to the* **rm_period** *directive.*
- *A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout specified.*
- *A running task issues a directive which deletes a message queue, a sema–phore, or a region on which the blocked task is waiting.*
- *A running task issues a* **t_restart** *directive for the blocked task.*
- *The running task, with its preemption mode enabled, may be made* **ready** *by issuing any of the directives that may unblock a task with a higher priority this directive maybe issued from the running task itself or from an ISR.*

A ready task occupies the **executing** state when it has control of the CPU. A task enters the **executing** state due to any of the following conditions:

- *The task is the highest priority ready task in the system.*
- *The running task blocks and the task is next in the scheduling queue. The task may be of equal priority as in round–robin scheduling or the task may possess the highest priority of the remaining ready tasks.*
- *The running task may reenable its preemption mode and a task exists in the ready queue that has a hiker priority than the running task.*
- *The running task lowers its own priority and another task is of higher priority as a result.*
- *The running task raises the priority of a task above its own and the running task is in preemption mode.*

XVII. RATE MONOTONIC MANAGER

A. Introduction

The **rate monotonic manager** provides facilities to implement tasks which execute in a periodic fashion. The directives provided by the rate monotonic manager are:

Name	Directive Description
rm_create	Create a rate monotonic timer
rm_cancel	Cancel a period
rm_delete	Delete a rate monotonic timer
rm_period	Conclude current/Start next period

B. Background

The rate monotonic manager provides facilities to manage the execution of periodic tasks. This manager was designed to support application designers who utilize the **Rate Monotonic Scheduling Algorithm** (RMS) to insure that their periodic tasks will meet their deadlines, even under transient overload conditions. Although designed for hard real–time systems, the services provided by the rate monotonic manager may be used by any application which requires periodic tasks.

1. Definitions

A **periodic task** is one which must be executed at a regular interval. The interval between successive iterations of the task is referred to as its period. Periodic tasks can be characterized by the length of their period and execution time. The period and execution time of a task can be used to determine the processor utilization for that task. **Processor utilization** is the percentage of processor time used and can be calculated on a per–task or system–wide basis. Typically, the task's worst–case execution time will be less than its period. For example, a periodic task's requirements may state that it should execute for 10 milliseconds every 100 milliseconds. Although the execution time may be the average, worst, or best case, the worst–case execution time is more appropriate for use when analyzing system behavior under transient overload conditions.

In contrast, an **aperiodic task** executes at irregular intervals and has only a soft deadline. In other words, the deadlines for aperiodic tasks are not rigid, but adequate response times are desirable. For example, an aperiodic task may process user input from a terminal.

Finally, a **sporadic task** is a aperiodic task with a hard deadline and minimum interarrival time. The **minimum interarrival** time is the minimum period of time which exists between successive iterations of the task. For example, a sporadic task could be used to process the pressing of a fire button on a joystick. The mechanical action of the fire button insures a minimum time period between successive activations, but the missile must be launched by a hard deadline.

2. Rate Monotonic Scheduling Algorithm

The **Rate Monotonic Scheduling Algorithm** (RMS) is important to real–time systems designers because it allows one to guarantee that a set of tasks is **schedulable**. A set of tasks is said to be **schedulable** if all of the tasks can meet their deadlines. RMS provides a set of rules which can be used to perform a **guaranteed schedulability analysis** for a task set. This analysis determines whether a task set is schedulable under worst–case conditions and emphasizes the predictability of the system's behavior. It has been proven that:

RMS is an optimal priority algorithm for scheduling independent, preemptible, periodic tasks on a single processor.

RMS is optimal in the sense that if a set of tasks can be scheduled by any static priority algorithm, then RMS will be able to schedule that task set. RMS bases it schedulability analysis on the processor utilization level below which all deadlines can be met.

RMS calls for the static assignment of task priorities based upon their period. The shorter a task's period, the higher its priority. For example, a task with a 1 mili–second period has higher priority than a task with a 100 millisecond period. If two tasks have the same period, than RMS does not distinguish between the tasks. However, RTEMS specifies that when given tasks of equal priority, the task which has been ready longest will execute first. RMS's priority assignment scheme does not provide one with exact numeric values for task priorities. For example, consider the following task set and priority assignments:

Task	Period (in milliseconds)	Priority
1	100	Low
2	50	Medium
3	50	Medium
4	25	High

RMS only calls for task 1 to have the lowest priority, task 4 to have the highest priority, and tasks 2 and 3 to have an equal priority between that of tasks 1 and 4. The actual **RTEMS** priorities assigned to the tasks must only adhere to those guidelines.

Many applications have tasks with both hard and soil deadlines. The tasks with hard deadlines are typically referred to as the **critical task set**, with the soil deadline tasks being the **non–critical task set**. The critical task set can be scheduled using RMS, with the non–critical tasks not executing under transient overload, by simply assigning priorities such that the lowest priority critical task (i.e., longest period) has a higher priority than the highest priority non–critical task. Although RMS may be used to assign priorities to the non–critical tasks, it is not necessary. In this instance, schedulability is only guaranteed for the critical task set.

3. Schedulability Analysis

RMS allows application designers to insure that tasks can meet all deadlines, even under transient overload, without knowing exactly when any given task will execute by applying proven schedulability analysis rules.

a. Assumptions

The schedulability analysis rules for RMS were developed based on the following assumptions:

- *The requests for all tasks for which hard deadlines exist are periodic, with a constant interval between requests.*
- *Each task must complete before the next request for it occurs.*
- *The task are independent in that a task does not depend on the initiation or completion of requests for other tasks.*
- *The execution time for each task without preemption or interruption is constant and does not vary.*
- *Any non–periodic tasks in the system are special. These tasks displace periodic tasks while executing and do not have hard, critical deadlines.*

Once the basic schedulability analysis is understood, some of the above assumptions can be relaxed and the side–effects accounted for.

b. Processor Utilization Rule

The **Processor Utilization Rule** requires that processor utilization be calculated based upon the period and execution time of each task. The fraction of processor time spent executing task **i** is Time[i]/Period[i]. The processor utilization can be calculated as follows:

```
Utilization = 0
    for i = 1 to max_tasks
                Utilization = Utilization + (Time[i]/Period[i])
```

To insure schedulability even under transient overload, the processor utilization must adhere to the following rule:

$$\text{Utilization} = \text{max\_tasks} * (2^{(1/\text{max\_tasks})} - 1)$$

As the number of tasks increases, the above formula approaches ln(2) for a worst–case utilization factor of approximately 0.693. Many tasks sets can be scheduled with a greater utilization factor. In fact, the average processor utilization threshold for a randomly generated task set is approximately 0.88.

c. Processor Utilization Rule Example

This example illustrates the application of the **Processor Utilization Rule** to an application with three critical periodic tasks. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	15	0.15
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for this task set is 0.73 which is below the upper bound of $3 * (2^{(1/3)} - 1)$, or 0.779, imposed by the **Processor Utilization Rule**. Therefore, this task set is guaranteed to be schedulable using RMS.

d. First Deadline Rule

If a given set of tasks do exceed the processor utilization upper limit imposed by the **Processor Utilization Rule**, they can still be guaranteed to meet all their deadlines by application of the **First Deadline Rule**. This rule can be stated as follows:

For a given set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times.

A key point with this rule is that **ALL** periodic tasks are assumed to start at the exact same instant in time. Although this assumption may seem to be invalid, **RTEMS** makes it quite easy to insure. By having a non–preemptible user initialization task, all application tasks, regardless of priority, can be created and started before the initialization deletes itself. This technique insures that all tasks begin to compete for execution time at the same instant – – when the user initialization task deletes itself.

e. First Deadline Rule Example

The **First Deadline Rule** can insure schedulability even when the **Processor Utilization Rule** fails. The example below is a modification of the Processor Utilization Rule example where task execution time has been increased from 15 to 25 units. The following table details the RMS priority, period, execution time, and processor utilization for each task:

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	25	0.25
2	Medium	200	50	0.25
3	Low	300	100	0.33

The total processor utilization for the modified task set is 0.83 which is above the upper bound of $3 * (2^{(1/3)} - 1)$, or 0.779, imposed by the **Processor Utilization Rule**. Therefore, this task set is not guaranteed to be schedulable using **RMS**. However, the **First Deadline Rule** can guarantee the schedulability of this task set. This rule calls for one to examine each occurrence of deadline until either all tasks have met their deadline or one task failed to meet its first deadline. The following table details the time of each deadline occurrence, the maximum number of times each task may have run, the total execution time, and whether all the deadlines have been met.

Deadline Time	Task 1	Task 2	Task 3	Total Execution Time	All Deadlines Met?
100	1	1	1	25 + 50 + 100 = 175	NO
200	2	1	1	50 +50 + 100 = 200	YES

The key to this analysis is to recognize when each task will execute. For example, at time 100, task 1 must have met its first deadline, but tasks 2 and 3 may also have begun execution. In this example, at time 100 tasks 1 and 2 have completed execution and thus have met their first deadline. Tasks 1 and 2 have used (25 + 50) = 75 time units, leaving (100 – 75) = 25 time units for task 3 to begin. Because task 3 takes 100 ticks to execute, it will not have completed execution at time 100. Thus, at time 100, all of the tasks except task 3 have met their first deadline.

At time 200, task 1 must have met its second deadline and task 2 its first deadline. As a result, of the first 200 time units, task 1 uses (2 * 25) = 50 and task 2 uses 50, leaving (200 – 100) time units for task 3. Task 3 requires 100 time units to execute, thus it will have completed execution at time 200. Thus, all of the tasks have met their first deadlines at time 200, and the task set is schedulable using the **First Deadline Rule**.

f. Relaxation of Assumptions

The assumptions used to develop the RMS schedulability rules are uncommon in most real–time systems. For example, it was assumed that tasks have constant unvarying execution time. It is possible to relax this assumption, simply by using the worst–case execution time of each task.

Another assumption is that the tasks are independent. This means that the tasks do not wait for one another or contend for resources. This assumption can be relaxed by accounting for the amount of time a task spends waiting to acquire resources. Similarly, each task's execution time must account for any I/O performed and any RTEMS directive calls.

In addition, the assumptions did not account for the time spent executing interrupt service routines. This can be accounted for by including all the processor utilization by interrupt service routines in the utilization calculation. Similarly, one should also account for the impact of delays in accessing local memory caused by direct memory access and other processors accessing local dual–ported memory.

The assumption that nonperiodic tasks are used only for initialization or failure–recovery can be relaxed by placing all periodic tasks in the critical task set which can be scheduled and analyzed using RMS. All nonperiodic tasks are placed in the non–critical task set. Although the critical task set can be guaranteed to execute even under transient overload, the non–critical task set is not guaranteed to execute.

In conclusion, the application designer must be fully cognizant of the system and its run–time behavior when performing schedulability analysis for a system using RMS. Every factor, both software and hardware, which impacts the execution time of each task must be accounted for in the schedulability analysis.

4. Further Reading

For more information on **Rate Monotonic Scheduling** and its schedulability analysis, the reader is referred to the following:

- C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment." Journal of the Association of Computing Machinery. January 1973. pp. 46–61.
- John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior." IEEE Real–Time Systems Symposium. 1989. pp. 166–171.
- Lui Sha and John Goodenough. "Real–Time Scheduling Theory and Ada." IEEE Computer. April 1990. pp. 53–62.
- Alan Burns. "Scheduling hard real–time systems: a review." Software Engineering Journal. May 1991. pp. 116–128.

C. Operations

1. Creating a Rate Monotonic Timer

The **rm_create** directive creates a rate monotonic timer which is to be used by the calling task to delineate a period. **RTEMS** allocates a **Timer Control Block** (TMCB) from the TMCB free list. This data structure is used by **RTEMS** to manage the newly created rate monotonic timer. **RTEMS** returns a unique timer ID to the application which is used by other rate monotonic manager directives to access this rate monotonic timer.

2. Manipulating a Period

The **rm_period** directive is used to establish and maintain a period utilizing a previously created rate monotonic timer. Once initiated by the **rm_period** directive, the period is said to run until it either expires or is reinitiated. The state of the rate monotonic timer results in one of the following scenarios:

- *If the rate monotonic timer is running, the calling task will be blocked for the remainder of the outstanding period and, upon completion of that period, the timer will be reinitiated with the specified* **period.**
- *If the rate monotonic timer is not currently running and has not expired, it is initiated with a length of* **period** *ticks and the calling task returns immediately*
- *If the rate monotonic timer has expired before the task invokes the* **rm_period** *directive, the timer will be initiated with a length of* **period** *ticks and the calling task returns immediately with a timeout error status.*

3. Obtaining a Period's Status

If the **rm_period** is invoked with a **period** of **STATUS** ticks, the current state of the specified rate monotonic timer will be returned. The following table details the relationship between the timer's status and the directive status code returned by **rm_period** directive:

Directive Status	Timer State
SUCCESSFUL	timer is running
E_TIMEOUT	timer has expired
E_NOTDEFINED	timer has never been Initiated

Obtaining the status of a rate monotonic timer does not alter the state or period of the timer.

4. Canceling a Period

The **rm_cancel** directive is used to stop the period maintained by the specified rate monotonic timer. The period is stopped and the rate monotonic timer can be reinitiated using the **rm_period** directive.

5. Deleting a Rate Monotonic Timer

The **rm_delete** directive is used to delete a rate monotonic timer. If the timer is running and has not expired, the period is automatically canceled. The rate monotonic timer's control block is returned to the TMCB free list when it is deleted. A rate monotonic timer can be deleted by a task other than the task which created the timer.

6. Examples

The following sections illustrate common uses of rate monotonic timers to construct periodic tasks.

a. Simple Periodic Task

This example consists of a single periodic task which, after initialization, executes every 100 clock ticks.

```
task Periodic_task ( )
{
        obj_id          rmid;
        dir_status      retval;

        rm_create ( &rmid );
        while ( 1 ) {
            retval = rm_period ( rmid, 100 );
            if ( retval == E_TIMEOUT )
                break;

        /* Perform some periodic actions */
}
/* missed period so delete timer and SELF */
rm_delete ( rmid );
t_delete ( SELF );
}
```

The above task creates a rate monotonic timer as part of its initiallzation. The first time the loop is executed, the **rm_period** directive will initiate the period for 100 ticks and return immediately. Subsequent invocations of the **rm_period** directive will result in the task blocking for the remainder of the 100 tick period. If, for any reason, the body of the

loop takes more than 100 ticks to execute, the **rm_period** directive will return the E_TIMEOUT status. If the above task misses its deadline, it will delete the rate monotonic timer and itself.

b. Task with Multiple Periods

This example consists of a single periodic task which, after initialization, performs two sets of actions every 100 clock ticks. The first set of actions is performed in the first forty clock ticks of every 100 clock ticks, while the second set of actions is performed between the fortieth and seventieth clock ticks. The last thirty clock ticks are not used by this task.

```
task Periodic_task ( )
{
    obj_id          rmid1, rmid2;
    dir_status      retval;

    rm_create ( &rmid1 );
    rm_create ( &rmid2 );
    while ( 1 ) {
        retval = rm_period ( rmid1, 100 );
        if ( retval == E_TIMEOUT )
            break;

    retval = rm_period ( rmid2, 40 )
    if ( retval == E_TIMEOUT )
        break

    /* Perform first set of actions between clock
     * ticks 0 to 39 of every 100 ticks.
    */

    retrval = rm_period ( rmid2, 30 );
    if ( retval = E_TIMEOUT )
        break;

    /* Perform second set of actions between clock
     * 40 and 69 of every 100 ticks.
    */

    /* Check to make sure we didn't miss
    * the rmid2 period.
    */

    retval = rm_period ( rmid2, STATUS );
    if ( retval == E_TIMEOUT )
        break;

    rm_cancel ( rmid2 )
}
/* missed period so delete timer and SELF */
rm_delete ( rmid1 );
rm_delete ( rmid2 );
t_delete ( SELF );
}
```

The above task creates two rate monotonic timers as part of its initialization. The first time the loop is executed, the rm_period directive will initiate the rmid1 period for 100 ticks and return immediately. Subsequent invocations of the rm_period directive for rm_period directive rmid1 will result in the task blocking for the remainder of the 100 tick period. The **rmid2** timer is used to control the execution time of the two sets of actions within each 100 tick period extableshed by **rmid 1.** The **rm_cancel(rmid2)** call is performed to insure that the **rmid2** period does not expire while the task is blocked on the **rmid1** period. If this cancel operation were not performed, every time the **rm_period(rmid1,40)** call is executed, except for the initial one , a directive status of **E_TIMEOUT** is returned. It is important to note that every time this call is made, the **rmid1** timer will be initiated immediately and the task will not block.

If, for any reason, the task misses any deadline, the **rm_period** directive will return the **E_TIMEOUT** directive status. If the above task misses its deadline, it will delete the rate monotonic timers and itself.

D. Directives

This section details the rate monotonic manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

1. **RM_CREATE – Create a rate monotonic timer**

CALLING SEQUENCE:

dir_status rm_create (&rmid)

INPUT: NONE

OUTPUT:

obj_id *rmid; /* id assigned to period */

DIRECTIVE STATUS CODES:

SUCCESSFUL	rate monotonic timer created successfully
E_TOOMANY	too many timers created

DESCRIPTION:

This directive creates a rate monotonic timer. The assigned rate monotonic and is returned in **rmid**. This id is used to access the timer with other rate monotonic manager directives. For control and maintenance of the rate monotonic timer, **RTEMS** allocates a TMCB from the local TMCB free pool and initializes it.

NOTES:

This directive will not cause the calling task to be preempted.

2. **RM_CANCEL – Cancel a period**

CALLING SEQUENCE:

dir_status rm_cancel (&rmid)

INPUT: NONE

OUTPUT:

objid *rmid; /* rate monotonic timer id */

DIRECTIVE STATUS CODES:

SUCCESSFUL	period canceled successfully
E_ID	invalid rate monotonic timer and
E_CALLED	rate monotonic timer not created by calling task

DESCRIPTION:

This directive cancels the rate monotonic timer **rmid**. This timer will be re-initiated by the next invocation of **rm_period** with rmid.

NOTES:

This directive will not cause the running task to be preempted.

The rate monotonic timer specified by rmid must have been created by the calling task.

3. **RM_DELETE – Delete a rate monotonic timer**

CALLING SEQUENCE:

dir_status rm_delete (rmid)

INPUT:

obj_id rmid; /* rate monotonic timer id */

OUTPUT: NONE

DIRECTIVE STATUS CODES:

SUCCESSFUL	period deleted successfully
E_ID	invalid rate monotonic timer id

DESCRIPTION:

This directive deletes the rate monotonic timer specified by **rmid**. If the timer is running, it is automatically canceled. The TMCB for the deleted timer is reclaimed by RTEMS.

NOTES:

This directive will not cause the running task to be preempted.

A rate monotonic timer can be deleted by a task other than the task which created the timer.

4. RM_PERIOD – Conclude current/Start next period

CALLING SEQUENCE:

dir_status rm_period (rmid, period)

INPUT:

```
obj_id    rmid;     /* rate monotonic timer id     */
interval  period;   /* length of period (in ticks) */
```

OUTPUT:

DIRECTIVE STATUS CODES:

SUCCESSFUL	period initiated successfully
E_ID	invalid rate monotonic timer id
E_CALLED	rate monotonic timer not created by calling task
E_NOTDEFINED	period has never been initiated
ETIMEOUT	period has expired

DESCRIPTION:

This directive initiates the rate monotonic timer **rmid** with a length of period ticks. If **rmid** is running, then the calling task will block for the remainder of the period before reinitiating the timer with the specified period. If **rmid** was not running (either expired or never initiated), the timer is immediately initiated id the directive returns immediately.

If invoked with a period of **STATUS** ticks, the current state of **rmid** will be returned. The directive status indicates the current state of the timer. This does not alter the state or period of the timer.

NOTES:

This directive will not cause the running task to be preempted.

XVIII. BOARD SUPPORT PACKAGES

A. Introduction

A **Board Support Package** (BSP) is a collection of user–provided facilities which interface **RTEMS** and a application with a specific hardware platform. These facilities may include hardware initialization, device drivers, user extensions, and a Multiprocessor Communications Interface (MPCI). However, a minimal BSP need only support processor reset and initialization and, if needed, a clock tick.

B. Reset and Initialization

An **RTEMS** based application is initiated or re–initiated when the processor is reset. This initialization code is responsible for preparing the target platform for the **RTEMS** application. Although the exact actions performed by the initialization code are highly processor and target dependent, the logical functionality of these actions are similar across a variety of processors and target platforms.

Normally, the application's initialization is performed at two separate times: before the call to **init_exec** (reset application initialization) and after **init_exec** in the user's initialization tasks (local and global application initialization). The order of the startup procedure is as follows:

1. Reset application initialization.
2. Call to **init_exec**
3. Local and global application initialization.

The reset application initialization code is executed first when the processor is reset. All of the hardware must be initialized to a quiescent state by this software before initializing **RTEMS**. When in quiescent state, devices do not generate any interrupts or require any servicing by the application. Some of the hardware components may be initialized in this code as well as any application initialization that does not involve calls to **RTEMS** directives.

The processor's Interrupt Vector Table which will be used by the application must be set to the required value by the reset application initialization code. Because interrupts are enabled automatically by **RTEMS** as part of the **init_exec** directive, the Interrupt Vector Table MUST be set before this directive is invoked to insure correct interrupt vectoring. The processor's Interrupt Vector Table must be accessible by **RTEMS** as it will be modified by the **i_catch** directive. The reset code which is executed before the call to **init_exec** has the following requirements:

- *Must not make any* **RTEMS** *directive calls.*
- *If the processor supports multiple privilege levels, must leave the processor in the most privileged, or supervisory, state.*
- *Must allocate a stack of at least* **MIN_STK_SIZE** *bytes and initialize the stack pointer for the* **init_exec** *directive.*
- *Must initialize the processor's Interrupt Vector Table.*
- *Must disable all maskable interrupts.*
- *If the processor supports a separate interrupt stab must locate the interrupt stack and initialize the interrupt stab pointer.*

The **init_exec** directive does not return to the initialization code, but causes the highest priority initialization task to begin execution. Initialization tasks are used to perform both local and global application initialization which is dependent on **RTEMS** facilities. The user initialization task facility is typically used to create the application's set of tasks.

1. Interrupt Stack Requirements

The worst–case stack usage by interrupt service routines must be taken into account when designing an application. If the processor supports interrupt nesting, the stack usage must include the deepest nest level. The worst–case stack usage must account for the following requirements:

- *Processor's interrupt stack frame*
- *Processor's subroutine call stack frame*
- **RTEMS** *system calls requiring up to* **MIN_STK_SIZE** *bytes*
- *Registers saved on stack*
- *Application subroutine calls*

The **RTEMS** constant **MIN_STZ_SIZE** includes all subroutine call stack frames and registers saved on the stack by **RTEMS**.

2. Processors with a Separate Interrupt Stack

Some processors support a separate stack for interrupts. When an interrupt is vectored and the interrupt is not nested, the processor will automatically switch from the current stack to the interrupt stack. The size of this stack is based solely on the worst–case stack usage by interrupt service routines.

The dedicated interrupt stack for the entire application is supplied and initialized by the reset and initialization code of the user's board support package. Since all ISRs use this stack, the stack size must take into account the worst case stack usage by any combination of nested ISRs.

3. Processors without a Separate Interrupt Stack

Some processors do not support a separate stack for interrupts. In this case, every task's stack must include enough space to handle the task's worst–case stack usage as well as the worst–case interrupt stack usage. This is necessary because the worst–case interrupt nesting could occur while any task is executing.

C. Device Drivers

Device drivers consist of control software for special peripheral devices and provide a logical interface for the application developer. The **RTEMS I/O manager** provides directives which allow applications to access these device drivers in a consistent fashion. A Board Support Package may include device drivers to access the hardware on the target platform. These devices typically include serial and parallel ports, counter/timer peripherals, real–time clocks, disk interfaces, and network controllers.

For more information on device drivers, refer to the **I/O Manager** chapter.

1. Clock Tick Device Driver

Most **RTEMS** applications will include a clock tick device driver which invokes the **tm_tick** directive at regular intervals. The clock tick is necessary if the application is to utilize timeslicing, the time and rate monotonic managers, or the timeout option on blocking directives.

The clock tick is usually provided as an interrupt from a counter/timer or a real–time clock device. When a counter/timer is used to provide the clock tick, the device is typically programmed to operate in continuous mode. This mode selection causes the device to automatically reload the initial count and continue the countdown without programmer intervention. This reduces the overhead required to manipulate the counter/timer in the clock tick ISR and increases the accuracy of tick occurrences. The initial count can be based on the **ms_tick** field in the **RTEMS Configuration Table**. An alternate approach is to set the initial count for one millisecond and have the ISR invoke **tm_tick** on the **ms_tick** boundaries.

It is important to note that the interval between clock ticks directly impacts the granularity of **RTEMS** timing operations. In addition, the frequency of clock ticks is an important factor in the overall level of system overhead. A high clock tick frequency results in less processor time being available for task execution due to the increased number of clock tick ISRs.

D. User Extensions

RTEMS allows the application developer to augment selected features by invoking user–supplied extension routines when the following system events occur:

- *Task creation*
- *Task initiation*
- *Task reinitiation*
- *Task deletion*
- *Task context switch*
- *Task exits*
- *Fatal error detection*

User extensions can be used to implement a wide variety of functions including execution profiling, non–standard coprocessor support, debug support, and error detection and recovery. For example, the context of a non–standard numeric coprocessor may be maintained via the user extensions. In this example, the task creation and deletion extensions are responsible for allocating and deallocating the context area, the task initiation and reinitiation extensions would be responsible for priming the context area, and the task context switch extension would save and restore the context of the device.

For more information on user extensions, refer to the **User Extensions** chapter.

E. Multiprocessor Communications Interface (MPCI)

RTEMS requires that an MPCI layer be provided when a multiple node application is developed. This MPCI layer must provide an efficient and reliable communications mechanism between the multiple nodes. Tasks on different nodes communicate and synchronize with one another via the MPCI. Each MPCI layer must be tailored to support the architecture of the target platform.

For more information on the MPCI, refer to the **Multiprocessing Manager** chapter.

1. Tightly–Coupled Systems

A tightly–coupled system is a multiprocessor configuration in which the processors communicate solely via shared global memory. The MPCI can simply place the **RTEMS** packets in the shared memory space. The two primary considerations when designing an MPCI for a tightly–coupled system are data consistency and informing another node of a packet.

The data consistency problem may be solved using atomic "test and set" operations to provide a "lock" in the shared memory. It is important to minimize the length of time any particular processor locks a shared data structure.

The problem of informing another node of a packet can be addressed using one of two techniques. The first technique is to use an interprocessor interrupt capability to cause an interrupt on the receiving node. This technique requires that special support hardware be provided by either the processor itself or the target platform. The second technique is to have a node poll for arrival of packets. The drawback to this technique is the overhead associated with polling.

2. Loosely–Coupled Systems

A loosely–coupled system is a multiprocessor configuration in which the processors communicate via some type of communications link which is not shared global memory. The MPCI sends the **RTEMS** packets across the communications link to the destination node. The characteristics of the communications link vary widely and have a significant impact on the MPCI layer. For example, the bandwidth of the communications link has an obvious impact on the maximum MPCI throughput.

The characteristics of a shared network, such as Ethernet, lend themselves to supporting an MPCI layer. These networks provide both the point–to–point and broadcast capabilities which are expected by **RTEMS**.

3. Systems with Mixed Coupling

A mixed–coupling system is a multiprocessor configuration in which the processors communicate via both shared memory and communications links. A unique characteristic of mixed–coupling systems is that a node may not have access to all communication methods. There may be multiple shared memory areas and communication links. Therefore, one of the primary functions of the MPCI layer is to efficiently route **RTEMS** packets between nodes. This routing may be based on numerous algorithms. In addition, the router may provide alternate communications paths in the event of an overload or a partial failure.

4. Heterogeneous Systems

Designing an MPCI layer for a heterogeneous system requires special considerations by the developer. **RTEMS** is designed to eliminate many of the problems associated with sharing data in a heterogeneous environment. The MPCI layer need only address the representation of the **unsigned32** data type.

For more information on supporting a heterogeneous system, refer the **Supporting Heterogeneous Environments** in the **Multiprocessing Manager** chapter.

XIX. USER EXTENSIONS

A. Introduction

RTEMS allows the application developer to augment selected features by invoking user–supplied extension routines when the following system events occur:

- *Task creation*
- *Task initiation*
- *Task reinitiation*
- *Task deletion*
- *Task context switch*
- *Task exits*
- *Fatal error detection*

These extensions are invoked as C functions with arguments that are appropriate to the system event. These functions are defined in the application's **User Extension Table which is included as part of the Configuration Table. All user extensions are optional and** RTEM places no naming restrictions on the user.

In addition, **RTEMS** provides for a user–defined data area to be linked to each task's control block. This data area is a extension of the TCB and can be used to store additional data required by one or more of the user's extension functions. It is also possible for a user extension to utilize the notepad locations associated with each task.

The sections that follow will contain a description of each extension. Each section will contain a example of the C calling sequence for the corresponding extension. The names given for the C function and its arguments are all defined by the user. The names used in the examples were arbitrarily chosen and impose no naming conventions on the user.

B. TCREATE Extension

The TCREATE extension directly corresponds to the **t_create** directive. If this extension is defined in the **Configuration Table** and a task is being created, then the extension routine will automatically be invoked by **RTEMS**. The extension should be prototyped as follows:

```
void    tcreate (curtcb, newtcb)
t_cb    *curtcb;
t_cb    *newtcb;
```

where **curtcb** is the pointer to the TCB for the currently executing task, and **newtcb** is the pointer to the TCB for the new task being created. This extension is invoked from thc **t_create** directive after **newtcb** has been completely initialized, but before it is placed on a ready TCB chain.

C. TSTART Extension

The TSTART extension directly corresponds to the **t_start** directive. If this extension is defined in the **Configuration Table** and a task is being started, then the extension routine will automatically be invoked by **RTEMS**. The extension should be prototyped as follows:

```
void tstart (curtcb, sttcb)
t_cb *curtcb;
t_cb *sttcb;
```

where **curtcb** is the pointer to the TCB for the currently executing task, and **sttcb** is the pointer to the TCB for the task being started. This extension is invoked from the **t_restart** directive after sttcb has been made ready to start execution, but before it is placed on a ready TCB chain.

D. TRESTART Extension

The TRESTART extension directly corresponds to the **t_restart** directive. If this extension is defined in the **Configuration Table** ad a task is being restarted, then the extension should be prototyped as follows:

```
void trestart (curtcb, sttcb)
t_cb *curtcb;
t_cb *sttcb;
```

where curtcb is the pointer to the TCB for the currently executing task, and **sttcb** is the pointer to the TCB for the task being restarted. This extension is invoked from the trestart directive after **sttcb** has been made ready to start execution, but before it is placed on a ready TCB chain.

E. TDELETE Extension

The TDELETE extension is associated with the tdelete directive. If this extension is defined in the **Configuration Table** and a task is being deleted, then the extension routine will automatically be invoked by RTEMS. The extension should be prototyped as follows:

```
void tdelete (curtcb, deltcb)
t_cb *curtcb;
t_cb ·deltcb;
```

where **curtcb** is the pointer to the TCB for the currently executing task, and **deltcb** is the pointer to the TCB for the task being deleted. This extension is invoked from the **t_delete** directive after the TCB has been removed from a ready TCB chain, but before all its resources including the TCB have been returned to their respective free pools. This extension should not call any **RTEMS** directives if a task is deleting itself (**curtcb** is equal to **deltcb**).

F. TSWITCH Extension

The TSWITCH extension corresponds to a task context switch. If this extension is defined in the **Configuration Table** and a task context switch is in progress, then the extension routine will automatically be invoked by **RTEMS**. The extension should be prototyped as follows:

```
void tswitch (curtcb, heirtcb)
t_cb *curtcb;
t_cb *heirtcb;
```

where **curtcb** is the pointer to the TCB for the task that is being swapped out, and **heirtcb** is the pointer to the TCB for the task being swapped in. This extension is invoked from **RTEMS'** dispatcher routine after the **curtcb** context has been saved, but before the **heirtcb** context has been restored. This extension should not call any **RTEMS** directives.

G. TASKEXITTED Error Extension

The TASKEXITTED error extension is invoked when a task exits the body of the starting procedure by either an implicit or explicit return statement. This user extension is prototyped as follows:

```
void taskexitted (curtcb)
t_cb *curtcb;
t_cb *heirtcb;
```

where **curtcb** is the pointer to the TCB for the currently executing task which has exitted. Although exiting of task is typically considered to be a fatal error, this extension allows recovery by either restarting or deleting the exiting task. If the user does not wish to recover, then a fatal error may be reported.

If the user does not provided a TASKEXITED error extension or the provided handler returns control to **RTEMS**, then the **RTEMS** default handler will be used. This default handler invokes the directive **k_fatal** with the E_EXITTED directive status.

H. FATAL Error Extension

The FATAL error extension is associated with the **k_fatal** directive. If this extension is defined in the **Configuration Table** is the **k_fatal** directive has been invoked, then this extension will be called. This extension should be prototyped as follows:

```
void k_fatal (errcode)
unsigned32 errcode;
```

where ***errcode*** *is the error code* passed to the **k_fatal** directive. This extension is invoked from the **k_fatal** directive.

If defined, the user's FATAL error extension is invoked before **RTEMS'** default fatal error routine is invoked and the processor is stopped. For example, this extension could be used to pass control to a debugger when a fatal error occurs. This extension should not call any **RTEMS** directives.

I. TCB Extension

The TCB extension is a pointer field in the TCB which can be set by the user to access an area of RAM. This allows an application to augment the TCB with user–defined information. For example, an application could implement task profiling by storing timing statistics in the TCB's extended memory area. When a task context switch is being executed, the TSWITCH extension could read a real–time clock to calculate how long the task being swapped out has run as well as timestamp the starting time for the task being swapped in.

If used, the extended memory area for the TCB should be allocated and the TCB extension pointer should be set at the time the task is created or started by either the TCREATE or TSTART extension. The application is responsible for managing this extended memory area for the TCBs. The memory may be reinitialized by the TRESTART extension and should be deallocated by the TDELETE extension when the task is deleted. Since the TCB extension buffers would most likely be of a fixed size, the **RTEMS** partition manager could be used to manage the application's extended memory area. The application could create a partition of fixed size TCB extension buffers and use the partition manager's allocation and deallocation directives to obtain and release the extension buffers.

XX. CONFIGURING A SYSTEM

A. Configuration Table

The **RTEMS Configuration Table** is used to tailor an application for its specific needs. For example, the user can configure the maximum number of task for this application. The address of the user–defined **Configuration Table** is passed as an argument to the **init_exec** directive, which EST be the first **RTEMS** directive called. The **RTEMS Configuration Table** is defined in a C structure. Each entry in the table is either two or four bytes in length. The structure is given here:

```
struct config_info {
    unsigned32      exec_ram;       /* RTEMS RAM Work Area*/
    unsigned32      ram_size;       /* RTEMS Work Area size   */
    unsigned16      max_tasks;      /* max number tasks       */
    unsigned16      max_semaphores; /* max number semaphores  */
    unsigned16      max_timers;     /· max number timers      */
    unsigned16      max_queues;     /* max number queues      */
    unsigned16      max_messages;   /* max number messages    */
    unsigned16      max_regions;    /* max number regions     */
    unsigned16      max_partitions; /* max number partitions  */
    unsigned16      max_dpmems;     /* max dp memory areas    */
    unsigned16      ms_tick;        /* ms in a tick           */
    unsigned16      tslice;         /* ticks in a timeslice   */
    unsigned32      num_itasks;     /* number of init tasks   */
    itask_table     *Itasks_tbl;    /* init task table        */
    unsigned32      num_devices;    /* number device drivers  */
    driver_table    *Drv_tbl;       /* driver table           */
    ext_table       *Ext_tbl;       /* extension table        */
    mp_table        *Mp_tbl;        /* MP config table
};
```

exe_ram is the starting address of the **RTEMS RAM Workspace**. This area contains items such as the various object control blocks (TCBs, QCBs, ...) and task stacks. If the address is not aligned on a four–word boundary, then RTEMS will invoke the fatal error handler during init_exec.

ram_size is the calculated size of the **RTEMS RAM Workspace**. The section **Sizing the RTEMS RAM Workspace** details how to arrive at this number.

max_tasks is the maximum number of tasks that can be concurrently active (created) in the system including initialization tasks.

max_semaphores is the maximum number of semaphores that can be concurrently active in the system.

max_timers is the maximum number of event and rate monotonic timers that can be concurrently active in the system.

max_queues is the maximum number of message queues that can be concurrently active in the system.

max_messages is the maximum number of messages that can be allocated to the application.

max_regions is the maximum number of regions that can be concurrently active in the system.

max_partitions is the maximum number of partitions that can be concurrently active in the system.

max_dpmems is the maximum number of dual–port memory areas that can be concurrently active in the system.

ms_tick is number of milliseconds per clock tick.

tstice is the number of clock ticks for a timeslice.

num_itasks is the number of initialization tasks configured. At least one initialization task must be configured.

Itasks_tbl is the address of the **Initialization Task Table**. This table contains the information needed to create and start each of the initialization tasks. The format of this table will be discussed below.

num_devices is the number of device drivers for the system. There should be the same number of entries in the **Device Driver Table**. If this field is zero, then the **Drv_tbl** entry should be **NULL_DRIVER_TABLE**.

Drv_tbl is the address of the **Device Driver Table**. This table contains the entry points for each device driver. If the **num_devices** field is zero, then this entry should be **NULL_DRIVER_TABLE**. The format of this table will be discussed below.

Ext_tbl is the address of the **User Extension Table**. This table contains the entry points for each user extension. If no user extensions are configured, then this entry should be *NULL_EXT_TABLE*. The format of this table will be discussed below.

Mp_tbl is the address of the **Multiprocessor Configuration Table**. This table contains information needed by **RTEMS** only when used in a multiprocessor configuration. This field must be **NULL_MP_TABLE** when **RTEMS** is used in a single processor configuration.

B. CPU Dependent Information Table

The **CPU Dependent Information Table** is used to describe processor dependent information required by **RTEMS**. This table is not required for all processors on which **RTEMS** is supported. The contents of this table are discussed in the **CPU Dependent Information Table** chapter of the **C Applications Supplement** document for a specific target processor.

C. Initialization Task Table

The **Initialization Task** Table is used to describe each of the user initialization tasks to the Initialization Manager. The table contains one entry for each initialization task the user wishes to create and start. The fields of the structure directly correspond to arguments to the **t_create** and **t_start** directives. The number of entries is found in the **num_itasks** entry in the **Configuration Table**. The format of each entry in the **Initialization Task Table** is defined in a C structure, and is given below:

```
struct itasks_info {
    obj_name     name;        /* task name          */
    unsigned32   stksize;     /* task stack size    */
    task_pri     priority;    /* task priority      */
    unsigned32   attributes;  /* task attributes    */
    task_ptr     entry;       /* task entry point   */
    unsigned32   mode;        /* task initial mode  */
    unsigned32   arg;         /* task argument      */
};
```

name is the name of this initialization task.

stksize is the size of the stack for this initialization task.

priority is the priority of this initialization task.

attributes is the attribute set used during creation of this initialization task.

entry is the address of the entry point of this initialization task.

mode is the initial execution mode of this initialization task.

arg is the initial argument for this initialization task.

A typical declaration for an **Initialization Task Table** might appear as follows:

```
itask_table Inittasks[2] = {
    { INIT1_NAME, 1024, 1, 0,
        init1, INTR (0) |NOPREEMPT, Init1_arg },
    ( INIT2_NAME, 1024, 1024, 1, 0,
        init2, INTR (0) |NOPREEMPT, Init2_arg }
};
```

D. Driver Address Table

The **Device Driver Table** is used to inform the I/O Manager of the set of entry points for each device driver configured in the system. The table contains one entry for each device driver required by the application. The number of entries is defined in the num_devices entry in the **Configuration Table**. The format of each entry in the **Device Driver Table** is defined in a C structure, and is given below:

```
struct driver_info {
    proc_ptr    init;       /* initialization procedure    */
    proc_ptr    open;       /* open request procedure      */
    proc_ptr    close;      /* close request procedure     */
    proc_tr     read;       /* read request procedure      */
    proc_ptr    write;      /* write request procedure     */
    proc_ptr    cntrl;      /* special request procedure   */
    unsigned32  reserved1;  /* reserved for RTEMS use      */
    unsigned32  reserved2;  /* reserved for RTEMS use      */
};
```

init is the address of the entry point called by **de_init** to initialize a device driver and its associated devices.

open is the address of the entry point called by **de_open**.

close is the address of the entry point called by **de_close.**

read is the address of the entry point called by **de_read.**

write is the address of the entry point called by **de_write.**

cntrl is the address of the entry point called by **de_cntrl.**

reserved1 is reserved for **RTEMS** use and should be set to **RESERVED.**

reserved2 is reserved for **RTEMS** use and should be set to **RESERVED.**

Driver entry points configured as **NULL_DRIVER** will always return a status code of **SUCCESSFUL**. No user code will be executed in this situation.

A typical declaration for a **Device Driver Table** might appear as follows:

```
driver_table Driver_table[2] = {
    { tty_open, tty_open, tty_close, tty_read,
      tty_write, tty_cntrl, RESERVED, RESERVED },
    { lp_open, lp_open, lp_close, NULL_DRIVER
      lp_write, lp_cntrl, RESERVED, RESERVED }
    };
```

More information regarding the construction and operation of device drivers is provided in the **I/O Manager** chapter.

E. User Extensions Table

The **User Extensions Table** is used to inform **RTEMS** of each of the optional user–supplied extensions. This table contains one entry for each possible extension. The entries are called at critical times in the life of a task. The format of each entry in the **User Extensions Table** is defined in a C structure, and is given below:

```
struct ext_info {
proc_ptr  tcreate;      /* tcreate user extension      */
proc_ptr  tstart;       /* tstart user extension       */
proc_ptr  trestart;     /* trestart user extension     */
proc_ptr  tdelete;      /* tdelete user extension      */
proc_ptr  tswitch;      /* tswitch user extension      */
proc_ptr  taskexitted;  /* task exit handler           */
proc_ptr  fatal;        /* fatal error handler         */
};
```

tcreate is the address of the user–supplied subroutine for the TCREATE extension. If this extension for task creation is defined, it is called from the **t_create** directive. A value of **NULL–EXTENSION** indicates that no extension is provided.

tstart is the address of the user–supplied subroutine for the TSTART extension. If this extension for task initiation is defined, it is called from the **t_start** directive. A value of **NULL–EXTENSION** indicates that no extension is provided.

trestart is the address of the user–supplied subroutine for the TRESTART extension. If this extension for task re–initiation is defined, it is called from the **t_restart** directive. A value of **NULL–EXTENSION** indicates that no extension is provided.

tdelete is the address of the user–supplied subroutine for the TDELETE extension. If this **RTEMS** extension for task deletion is defined, it is called from the **t_delete** directive. A value of **NULL–EXTENSION** indicates that no extension is provided.

tswitch is the address of the user–supplied subroutine for the task context switch extension. This subroutine is called from **RTEMS'** dispatcher after the current task has been swapped out but before the new task has been swapped in. A value of **NULL–EXTENSION** indicates that no extension is provided. As this routine is invoked alter saving the current task's context and before restoring the heir task's context, it is not necessary for this routine to save and restore any registers.

taskexitted is the address of the user–supplied subroutine which is invoked when a task exits. This procedure is responsible for some action which will allow the system to continue execution (i.e. delete or restart the task) or to terminate with a fatal error. If this field is set to **NULL_EXTENSION**, the default **RTEMS** taskexitted handler will be invoked.

fatal is the address of the user–supplied subroutine for the FATAL extension. This **RTEMS** extension of fatal error handling is called from the **k_fatal** directive. If the user's fatal error handler returns or if this entry is **NULL–EXTENSION** then the default **RTEMS** fatal error handler will be executed.

A typical declaration for a **User Extension Table** which defines the TCREATE, TDELETE, TSWITCH, and FATAL extension might appear as follows:

```
ext_table User_extensions = {
    tcreate_ext, NULL_EXTENSION, NULL_EXTENSION,
    tdelete_ext, tswitch_ext,
    NULL_EXTENSION, fatal_ext
};
```

More information regarding the user extensions is provided in the **User Extensions** chapter.

F. Multiprocessor Configuration Table

The **Multiprocessor Configuration Table** contains information needed when using **RTEMS** in a multiprocessor configuration. Many of the details associated with configuring a multiprocessor system are dependent on the multiprocessor communications layer provided by the user. The address of the **Multiprocessor Configuration Table** should be placed in the **Mp_tbl** entry in the primary **Configuration Table**. Further details regarding many of the entries in the Multiprocessor Configuration Table will be provided in the **Multiprocessing** chapter. The format of the **Multiprocessor Configuration Table** is defined in a C structure, and is given below:

```
struct mp_info {
    unsigned16    node;         /* local node number          */
    unsigned16    max_nodes;    /* number nodes in system     */
    unsigned32    max_gobjects; /* max global objects         */
    unsigned32    max_proxies;  /* max proxies                */
    mpci_table    *Mpci_tbl;    /* MPCI table
};
```

node is a unique processor identifier and is used in routing messages between nodes in a multiprocessor configuration. Each processor must have a unique node number. **RTEMS** assumes that node numbers start at one and increase sequentially. This assumption can be used to advantage by the user–supplied MPCI layer. Typically, this requirement is made when the node numbers are used to calculate the address of inter–processor communication links. Zero should be avoided as a node number because some MPCI layers use node zero to represent broadcasted packets. Thus, it is recommended that node numbers start at one and increase sequentially.

max_nodes is the number of processor nodes in the system.

max_gobjects is the maximum number of global objects which can exist at any given moment in the entire system. If this parameter is not the same on all

nodes in the system, then a fatal error is generated to inform the user that the system is inconsistent.

max_proxies is the maximum number of proxies which can exist at any given moment on this particular node. A **proxy** is a substitute task control block which represent a task residing on a remote node when that task blocks on a remote object. Proxies are used in situations in which delayed interaction is required with a remote node.

Mpci_tbl is the address of the **Multiprocessor Communications Interface Table.** This table contains the entry points of user–provided functions which constitute the multiprocessor communications layer. This table must be provided in multiprocessor configurations with all entries configured. The format of this table and details regarding its entries can be found in the next section.

G. Multiprocessor Communications interface Table

The format of this table is defined in a C structure, and is given below:

```
struct mpci_info {
    proc_ptr  init;      /* initialization procedure     */
    proc_ptr  getpkt;    /* get packet procedure         */
    proc_ptr  retpkt;    /* return packet procedure      */
    proc_ptr  send;      /* packet send procedure        */
    proc_ptr  receive;   /* packet receive procedure     */
};
```

init is the address of the entry point for the initialization procedure of the user supplied multiprocessor communications layer.

getpkt is the address of the entry point for the procedure called by **RTEMS** to obtain a packet from the user supplied multiprocessor communications layer.

retpkt is the address of the entry point for the procedure called by **RTEMS** to return a packet to the user supplied multiprocessor communications layer.

send is the address of the entry point for the procedure called by **RTEMS** to send an envelope to another node. This procedure is part of the user supplied multiprocessor communications layer.

receive is the address of the entry point for the procedure called by **RTEMS** to retrieve an envelope containing a message from another node. This procedure is part of the user supplied multiprocessor communications layer.

More information regarding the required functionality of these entry points is provided in the **Multiprocessor** chapter.

H. Determining Memory Requirements

Since memory is a critical resource in many real–time embedded systems, **RTEMS** was specifically designed to allow unused managers to be excluded from the run–time environment. This allows the application designer the flexibility to tailor **RTEMS** to most efficiently meet system requirements while still satisfy even the most stringent memory constraints. As result, the size of the **RTEMS** executive is application dependent. The **Memory Requirements** chapter of the **C Applications Supplement** document for a specific target processor provides a worksheet for calculating the memory requirements of a custom **RTEMS** run–time environment. To insure that enough memory is allocated for future versions of **RTEMS**, the application designer should round these memory requirements up. The following managers may be optionally excluded:

- *signal*
- *partition*
- *region*
- *time*
- *dual ported memory*
- *semaphore*
- *I/O*
- *message*
- *event*
- *rate monotonic*
- *multiprocessing*

RTEMS based applications must somehow provide memory for **RTEMS'** code and data space. Although **RTEMS'** data space must be in RAM, its code space can be located in either ROM or RAM. In addition, the user must allocate RAM for the RTEMS **RAM Workspace**. The size of this area is application dependent and can be calculated using the formula provided in the **Memory Requirements** chapter of the **C Applications Supplement** document for a specific target processor.

All **RTEMS** data variables and routine names used by **RTEMS** begin with the underscore (_) character followed by an upper–case letter. If **RTEMS** is linked with an application, then the application code should **NOT** contain any symbols which begin with the underscore character and an upper–case letter to avoid any naming conflicts. All **RTEMS** directive names should be treated as reserved words.

1. Sizing the RTEMS RAM Workspace

The **RTEMS RAM Workspace** is a user–specified block of memory reserved for use by **RTEMS**. The application should **NOT** modify this memory after it is cleared by the board support package. This area consists primarily of the **RTEMS** data structures whose exact size depends many values specified in the **Configuration Table**. In addition, task stacks and floating point context areas are dynamically allocated from the **RTEMS RAM Workspace**.

The starting address of the **RTEMS RAM Workspace** must be aligned on a four–byte boundary. Failure to properly align the workspace area will result in the *k_fatal* directive being invoked with the **E_ADDRESS** error code.

A worksheet is provided in the **Memory Requirements** chapter of the **C Applications Supplement** document for a specific target processor to assist the user m calculating the minimum size of the **RTEMS RAM Workspace** for this application. The value calculated with this worksheet is the minimum value that should be specified as the **ram_size** param-

eter of the **Configuration Table**. The user is cautioned that future versions of **RTEMS** may not have the same memory requirements per object. Although the value calculated is sufficient for a particular target processor and release of **RTEMS**, the user is advised to allocate somewhat more memory than the worksheet recommends to insure compatibility with future releases for a specific target processor and other target processors. Failure to provide enough space in the **RTEMS RAM Workspace** will result in the **k_fatal** directive being invoked with the E_UNSATISFIED error code.

XXI. MULTIPROCESSING MANAGER

A. Introduction

In multiprocessor real–time systems, new requirements, such as sharing data and global resources between processors, are introduced. This requires an efficient and reliable communications vehicle which allows all processors to communicate with each other as necessary. In addition, the verifications of multiple processors affect each and every characteristic of a real–time system, almost always making them more complicated.

RTEMS addresses these issues by providing simple and flexible real–time multiprocessing capabilities. The executive easily lends itself to both tightly–coupled and loosely–coupled configurations of the target system hardware. In addition, **RTEMS** supports both homogeneous and heterogeneous target environments.

A major design goal of the **RTEMS** executive was to transcend the physical boundaries of the target hardware configuration. This goal is achieved by presenting to the application software a logical view of the target system where the boundaries between processor nodes are transparent. As a result, the application developer may designate objects such as tasks, queues, events, signals, semaphores, and memory blocks as global objects. These global objects may then be accessed by any task regardless of which processors the object and the accessing task may reside. **RTEMS** automatically determines that the object being accessed resides on another processor and performs the actions required to access the desired object. Simply stated, **RTEMS** allows the entire system, both hardware and software, to be viewed logically as a single system.

B. Background

RTEMS makes no assumptions regarding the connection media or the topology of a multiprocessor system. The tasks which compose a particular application can be spread among several processors. The application tasks can interact using a subset of the **RTEMS** directives as if they were on the same processor. These directives allow application tasks to exchange data, communicate, and synchronize regardless of which processor they reside upon.

The **RTEMS** multiprocessor execution model is multiple instruction streams with multiple data streams (MIMD). This execution model has each of the processors executing code independent of the other processors. Because of this parallelism, the application designer can more easily guarantee deterministic behavior.

By supporting heterogeneous environments, **RTEMS** allows the systems designer to select the most efficient processor for each subsystem of the application. Configuring **RTEMS** for a heterogeneous environment is no more difficult than for a homogeneous one. In keeping with **RTEMS** philosophy of providing transparent physical node boundaries, the minimal heterogeneous processing required is isolated in the MPCI layer.

1. Nodes

A processor in a **RTEMS** system is referred to as a node. Each node is assigned a unique non–zero node number by the application designer. **RTEMS** assumes that node numbers are assigned consecutively from one to **max_nodes**. The node number, node, and the maximum number of nodes, **max_nodes**, in a system are found in the **Multiprocessor Configuration Table**. The **max_nodes** field and the number of global objects, **num_objects**, is required to be the same on all nodes in a system.

The node number is used by **RTEMS** to identify each node when performing remote operations. Thus, the **Multiprocessor Communications Interface Layer** (MPCI) must be able to route messages based on the node number.

2. Global Objects

All **RTEMS** objects which are created with the **GLOBAL** option will be known on all other nodes. Global objects can be referenced from any node in the system, although certain directive specific restrictions (e.g. cannot delete a remote object) may apply. A task does not have to be global to perform operations involving remote objects. The distribution of tasks to processors is performed during the application design phase. Dynamic task relocation is not supported by **RTEMS**.

3. Global Object Table

Every node in a multiprocessor system maintains two tables containing object information: a local object table and a global object table. The local object table on each node is unique and contains information for both local and global objects created on this node. The global object table contains information regarding all global objects in the system and therefore, is the same on every node.

Since each node must maintain an identical copy of the global object table, the maximum number of entries in each copy is determined by the **num_objects** parameter in the **Multiprocessor Configuration Table**. This parameter, as well as the **max_nodes** parameter, is required to be the same on all nodes. To maintain consistency among the table copies, every node in the system must be informed of the creation or deletion of a global object.

4. Remote Operations

When an application performs an operation on a remote global object, **RTEMS** must generate a Remote Request (RQ) message and send it to the appropriate node. After completing the requested operation, the remote node will build a Remote Response (RR) message and send it to the originating node. Messages generated as a side-effect of a directive (such as deleting a global task) are known as Remote Processes (RP) and do not require the receiving node to respond

Other than taking slightly longer to execute directives on remote objects, the application is normally unaware of the location of the objects it acts upon. The exact amount of overhead required for a remote operation is dependent on the media connecting the nodes and, to a lesser degree, the efficiency of the user-provided MPCI routines.

The following shows the typical transaction sequence during a remote application:

(1) The application issues a directive accessing a remote global object.

(2) **RTEMS** determines the node on which the object resides.

(3) **RTEMS** calls the user-provided MPCI routine GETPKT to obtain a packet in which to build a RQ message.

(4) After building a message packet, **RTEMS** calls the user provided MPCI routine SEND to transmit the packet to the node on which the object resides (referred to as the destination node).

(5) The calling task is blocked until the RR message arrives, and control of the processor is transferred to another task.

(6) The MPCI layer on the destination node senses the arrival of a packet (commonly in an ISR), and calls the **RTEMS mp_announce** directive. This directive readies the **Multiprocessing Server**.

(7) The **Multiprocessing Server** calls the user–provided MPCI routine RECEIVE, performs the requested operation, builds an RR message, and returns it to the originating node.

(8) The MPCI layer on the originating node senses the arrival of a packet (typically via an interrupt), and calls the **RTEMS mp_announce** directive. This directive readies the **Multiprocessing Server**.

(9) The **Multiprocessing Server** calls the user–provided MPCI routine RECEIVE, readies the original requesting task, and blocks until another packet arrives. Control is transferred to the original task which then completes processing the directive.

If an uncorrectable error occurs in the user–provided MPCI layer, the fatal error handler should be invoked. **RTEMS** assumes the reliable transmission and reception of messages by the MPCI and makes no attempt to detect or correct errors.

5. Proxies

A **proxy** is an **RTEMS** data structure which resides on a remote node and is used to represent a task which must block as part of a remote operation. This action can occur as part of the **sm_p** and **q_receive** directives. If the object were local, the task's control block would be available for modification to indicate it was pending a message or semaphore. However, the task's control block resides only on the same node as the task. In this case, the remote node must allocate a proxy to represent the task until it can be readied.

The maximum number of proxies is defined in the **Multiprocessor Configuration Table**. Each node in a multiprocessor system may require a different number of proxies to be configured. The distribution of proxy control blocks is application dependent and is different from the distribution of tasks.

6. Multiprocessor Configuration Table

The **Multiprocessor Configuration** Table contains information needed by **RTEMS** when used in a multiprocessor system. This table is discussed in detail in a section in the previous chapter, **Multiprocessor Configuration Table**.

C. Multiprocessor Communications Interface Layer

The **Multiprocessor Communications Interface Layer** (MPCI) is a set of user–provided procedures which enable the nodes in a multiprocessor system to communicate with one another. These routines are invoked by **RTEMS** at various times in the generation and processing of remote requests. Interrupts are enabled when an MPCI procedure is invoked. It is assumed that if the execution mode and/or interrupt level are altered by the MPCI layer, that they will be restored prior to returning to **RTEMS**.

The MPCI layer is responsible for managing a pool of buffers called **packets** and for sending these packets between system nodes. Packet buffers contain the messages sent

between the nodes. Typically, the MPCI layer will encapsulate the packet within an envelope which contains the information needed by the MPCI layer. The number of packets available is dependent on the MPCI layer implementation.

The entry points to the routines in the user's MPCI layer should be placed in the **Multiprocessor Communications Interface Table**. The user must provide entry points for each of the following table entries in a multiprocessor system:

init	initialize the MPCI
getpkt	obtain a packet buffer
retpkt	return a packet buffer
send	send a packet to another node
receive	called to get an arrived packet

A packet is sent by **RTEMS** in each of the following situations:

- *an RQ is generated on an originating node;*
- *an RR is generated on a destination node;*
- *a global object is created;*
- *a global object is deleted;*
- *a local task blocked on a remote object is deleted;*
- *during system initialization to check for system consistency.*

The arrival of a packet at a node may generate an interrupt. If it does not, the real-time clock ISR can check for the arrival of a packet. In any case, the **mp_announce** directive must be called to announce the arrival of a packet. After exiting the ISR, control will be passed to the **Multiprocessing Server** to process the packet. The **Multiprocessing Server** will call the **getpkt** entry to obtain a packet buffer and the **receive** entry to copy the message into the buffer obtained.

1. INIT

The INIT component of the user-provided MPCI layer is called as part of the **init_exec** directive to initialize the MPCI layer and associated hardware. It is invoked immediately after all of the device drivers have been initialized. This component should be prototyped as follows:

```
void init (conf_tbl)
config_table *conf_tbl;
```

where **conf_tbl** is the address of the user's **Configuration Table**. Operations on global objects cannot be performed until this component is invoked. The INIT component is invoked only once in the life of any system. If the MPCI layer cannot be successfully initialized, the fatal error manager should be invoked.

One of the primary functions of the MPCI layer is to provide the executive with packet buffers. The INIT routine must create and initialize a pool of packet buffers. There must be enough packet buffers so **RTEMS** can obtain one whenever needed.

2. GETPKT

The GETPKT component of the user–provided MPCI layer is called when RTEMS must obtain a packet buffer to send or broadcast a message. This component should be prototyped as follows:

```
void getpkt (pkt)
unsigned8 **pkt;
```

where **pkt** is the address of a pointer to a packet. This routine always succeeds and, upon return, **pkt** will contain the address of a packet. If for any reason, a packet cannot be successfully obtained, then the fatal error manager should be invoked.

RTEMS has been optimized to avoid the need for obtaining a packet each time a message is sent or broadcast. For example, RTEMS sends response messages (RR) back to the originator in the same packet in which the request message (RQ) arrived.

3. RETPKT

The RETPKT component of the user–provided MPCI layer is called when RTEMS needs to release a packet to the free packet buffer pool. This component should be prototyped as follows:

```
void retpkt (pkt)
unsigned8 *pkt;
```

where **pkt** is the address of a packet. If the packet cannot be successfully returned, the fatal error manager should be invoked.

4. RECEIVE

The RECEIVE component of the user–provided MPCI layer is called when RTEMS needs to obtain a packet which has previously arrived. This component should be prototyped as follows:

```
void receive (pkt)
unsigned8 **pkt;
```

where pkt is a pointer to the address of a packet to place the message from another node. If a message is available, then **pkt** will contain the address of the message from another node. If no messages are available, this entry **pkt** should contain **NULL_PACKET.**

5. SEND

The SEND component of the user–provided MPCI layer is called when RTEMS needs to send a packet containing a message to another node. This component should be prototyped as follows:

```
void send ( node, pkt, pktlength )
unsigned32    node;
unsigned8     *pkt;
unsigned32    pkt_length;
```

where **node** is the node number of the destination, **pkt** is the address of a packet which containing a message, and **pkt_length** is the length of the message in bytes. If the packet cannot be successfully sent, the fatal error manager should be invoked.

If **node** is set to zero, the packet is to be broadcasted to all other nodes in the system. Although some MPCI layers will be built upon hardware which support a broadcast mechanism, others may be required to generate a copy of the packet for each node in the system.

May MPCI layers use the **pkt_length** to avoid sending unnecessary data. This is especially useful if the media connecting the nodes is relatively slow.

6. Supporting Heterogeneous Environments

Developing an MPCI layer for a heterogeneous system requires a thorough understanding of the differences between the processors which comprise the system. One difficult problem is the varying data representation schemes used by different processor types. The most pervasive data representation problem is the order of the bytes which compose a data entity. Processors which place the least significant byte at the smallest address are classified as little endian processors. Little endian byte–ordering is shown below:

Conversely, processors which place the most significant byte at the smallest address are classified as big endian processors. Big endian byte–ordering is shown below:

Unfortunately, sharing a data structure between big endian and little endian processors requires translation into a common endian format. An application designer typically chooses the common endian format to the conversion overhead.

Another issue in the design of shared data structures is the alignment of data structure elements. Alignment is both processor and compiler implementation dependent. For example, some processors allow data elements to begin on any address boundary, while others impose restrictions. Common restrictions are that data elements must begin on either an even address or on a long word boundary. Violation of these restrictions may cause an exception or impose a performance penalty.

Other issues which commonly impact the design of shared data structures include the representation of floating point numbers, bit fields, decimal data, and character strings. In addition, the representation method for negative integers could be one's or two's complement. These factors combine to increase the complexity of designing and manipulating data structures shared between processors.

RTEMS addressed these issues in the design of the packets used to communicate between nodes. The **RTEMS** packet format is designed to allow the MPCI layer to perform all necessary conversion without burdening the developer with the details of the **RTEMS** packet format. As a result, the MPCI layer must be aware of the following:

- *All packets must begin on a long–word boundary.*
- *Packets are composed of both* **RTEMS** *and application data. All* **RTEMS** *data is unsigned32 and is located in the first* **MIN_HETERO_CONV** *unsigned32's of the packet*
- *The* **RTEMS** *data component of the packet must be in native endian format. Endian conversion may be performed by either the sending or receiving MPCI layer.*
- **RTEMS** *makes no assumptions regarding the application data component of the packet.*

D. Operations

1. Announcing a Packet

The **mp_announce** directive is called by the MPCI layer to inform **RTEMS** that a packet has arrived from another node. This directive can be called from an interrupt service routine or from within a polling routine.

E. Directives

This section details the additional directives required to support **RTEMS** in a multiprocessor configuration. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

MP_ANNOUNCE–Announce the arrival of a packet

CALLING SEQUENCE:

void mp_announce ( )

INPUT: NONE

OUTPUT: NONE

DIRECTIVE STATUS CODES: NONE

DESCRIPTION:

This directive informs **RTEMS** that a multiprocessing communications packet has arrived from another node. This directive is called by the user–provided MPCI, and is only used in multiprocessor configurations.

NOTES:

This directive is typically called from an ISR.

This directive will almost certainly cause the calling task to be preempted.

This directive does not generate activity on remote nodes.

APPENDIX A
DIRECTIVE STATUS CODES

A

Directive Status Codes

CONSTANT	CODE	DESCRIPTION
SUCCESSFUL	0	successful completion
E_EXITTED	1	returned from a task
E_NOMP	2	multiprocessing not configured
E_NAME	3	invalid object name
E_ID	4	invalid object id
E_TOOMANY	5	too many
E_TIMEOUT	6	timed out waiting
E_DELETE	7	object was deleted while waiting
E_SIZE	8	invalid specified size
E_ADDRESS	9	invalid address specified
E_NUMBER	10	number was invalid
E_NOTDEFINED	11	item not initialized
E_INUSE	12	resources outstanding
E_UNSATISFIED	13	request not satisfied
E_STATE	14	task is in wrong state
E_ALREADY	15	task already in state
E_SELF	16	illegal for calling task
E_REMOTE	17	illegal for remote object
E_CALLED	18	invalid environment
E_PRIORITY	19	invalid task priority
E_CLOCK	20	invalid time buffer
E_NODE	21	invalid node id
E_NOTCONFIGURED	22	directive not configured
E_NOTIMPLEMENTED	23	directive not implemented

APPENDIX B
EXAMPLE APPLICATION

B

Example Application

```
/*  example.c
 *
 *  This file contains an example of a simple RTEMS
 *  application.  It contains a Configuration Table, a
 *  user initialization task, and a simple task.
 *
 *  This example assumes that a board support package exists
 *  and invokes the in_executive() directive.  The board
 *  support package also provides the function Task_exitted()
 *  for the task exitted user-supplied routine.
 */

#include "rtems.h"

task init_task();

struct itasks_info init_task = {
  { 0x61626300,          /* init task name  "ABC"        */
    1024,                /* init task stack size         */
    1,                   /* init task priority           */
    DEFAULTS,            /* init task attributes         */
    init_task,           /* init task entry point        */
    TSLICE,              /* init task initial mode       */
    0,                   /* init task argument           */
  }
};

struct config_tbl Config_tbl = {
  0x0f0000,              /* exective RAM work area       */
  65536,                 /* exective RAM size            */
  2,                     /* maximum tasks                */
  0,                     /* maximum semaphores           */
  0,                     /* maximum timers               */
  0,                     /* maximum message queues       */
  0,                     /* maximum messages             */
  0,                     /* maximum regions              */
  0,                     /* maximum partitions           */
  0,                     /* maximum dp memory areas      */
  10,                    /* number of ms in a tick       */
  1,                     /* num of ticks in a timeslice  */
```

```
  1,                    /* number of user init tasks    */
  init_task_tbl,        /* user init task(s) table      */
  0,                    /* number of device drivers     */
  NULL_DRVER_TABLE,     /* ptr to driver address table  */
  NULL_EXT_TABLE,       /* ptr to extension table       */
  NULL_MP_TABLE,        /* ptr to MP config table       */
};

task user_application();

#define USER_APP_NAME  1  /* any 32-bit name; unique helps */

task init_task()
{
  obj_id tid;

  /* example assumes SUCCESSFUL return value */
  t_create( USER_APP_NAME, 1, 1024, NOPREEMPT, FP, &tid );
  t_start( app_tid, user_application, 0 );
  t_delete( SELF );
}

task user_application()
{
  /* application specific initialization goes here */

  while ( 1 ) {              /* infinite loop */
    /*  APPLICATION CODE GOES HERE
     *
     *  This code will typically include at least one
     *  directive which causes the calling task to
     *  give up the processor.
     */
  }

}
```

APPENDIX C
GLOSSARY

C

Glossary

active	A term used to describe an object which has been created by an application.
aperiodic task	A task which must execute only at irregular intervals and has only a soft deadline.
application	In this document, software which makes use of RTEMS.
ASR	see Asynchronous Signal Routine.
asynchronous	Not related in order or timing to other occurrences in the system.
Asynchronous Signal Routine	Similar to a hardware interrupt except that it is associated with a task and is run in the context of a task. The directives provided by the signal manager are used to service signals.
awakened	A term used to describe a task that has been unblocked and may be scheduled to the CPU.
big endian	A data representation scheme in which the bytes composing a numeric value are arranged such that the most significant byte is at the lowest address.
bit-mapped	A data encoding scheme in which each bit in a variable is used to represent something different. This makes for compact data representation.
block	A physically contiguous area of memory.

blocked	The task state entered by a task which has been previously started and cannot continue execution until the reason for waiting has been satisfied.
broadcast	To simultaneously send a message to a logical set of destinations.
BSP	see Board Support Package.
Board Support Package	A collection of device initialization and control routines specific to a particular type of board or collection of boards.
buffer	A fixed length block of memory allocated from a partition.
calling convention	The processor and compiler dependent rules which define the mechanism used to invoke subroutines in a high-level language. These rules define the passing of arguments, the call and return mechanism, and the register set which must be preserved.
Central Processing Unit	This term is equivalent to the terms processor and microprocessor.
chain	A data structure which allows for efficient dynamic addition and removal of elements. It differs from an array in that it is not limited to a predefined size.
coalesce	The process of merging adjacent holes into a single larger hole. Sometimes this process is referred to as garbage collection.
Configuration Table	A table which contains information used to tailor RTEMS for a particular application.
context	All of the processor registers and operating system data structures associated with a task.
context switch	Alternate term for task switch. Taking control of the processor from one task and transferring it to another task.
control block	A data structure used by the executive to define and control an object.
core	When used in this manual, this term refers to the internal executive utility functions. In the interest of application

	portability, the core of the executive should not be used directly by applications.
CPU	An acronym for Central Processing Unit.
critical section	A section of code which must be executed indivisibly.
CRT	An acronym for Cathode Ray Tube. Normally used in reference to the man-machine interface.
deadline	A fixed time limit by which a task must have completed a set of actions. Beyond this point, the results are of reduced value and may even be considered useless or harmful.
device	A peripheral used by the application that requires special operation software. See also device driver.
device driver	Control software for special peripheral devices used by the application.
directives	RTEMS' provided routines that provide support mechanisms for real-time applications.
dispatch	The act of loading a task's context onto the CPU and transferring control of the CPU to that task.
dormant	The state entered by a task after it is created and before it has been started.
Driver Address Table	A table which contains the entry points for each of the configured device drivers.
dual-ported	A term used to describe memory which can be accessed at two different addresses.
embedded	An application that is delivered as a hidden part of a larger system. For example, the software in a fuel-injection control system is an embedded application found in many late-model automobiles.
envelope	A buffer provided by the MPCI layer to RTEMS which is used to pass messages between nodes in a multiprocessor system. It typically contains routing information needed by the MPCI. The contents of an envelope are referred to as a packet.

entry point	The address at which a function or task begins to execute. In C, the symbolic entry point of a function is the function's name.
events	A method for task communication and synchronization. The directives provided by the event manager are used to service events.
exception	A synonym for interrupt.
executing	The task state entered by a task after it has been given control of the CPU.
executive	In this document, this term is used to referred to RTEMS. Commonly, an executive is a small real-time operating system used in embedded systems.
exported	An object known by all nodes in a multiprocessor system. An object created with the GLOBAL attribute set will be exported.
external address	The address used to access dual-ported memory by all the nodes in a system which do not own the memory.
FIFO	An acronym for First In First Out.
First In First Out	A discipline for manipulating entries in a data structure.
floating point coprocessor	A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.
freed	A resource that has been released by the application to RTEMS.
global	An object that has been created with the GLOBAL attribute set and exported to all nodes in a multiprocessor system.
handler	The equivalent of a manager, except that it is internal to RTEMS and forms part of the core. A handler is a collection of routines which provide a related set of functions. For example, there is a handler used by RTEMS to manage all objects.

hard real-time	A real-time system in which a missed deadline causes the worked performed to have no value or to result in a catastrophic effect on the integrity of the system.
heap	A data structure used to dynamically allocate and deallocate variable sized blocks of memory.
heterogeneous	A multiprocessor computer system composed of dissimilar processors.
homogeneous	A multiprocessor computer system composed of one type of processor.
ID	An RTEMS assigned identification tag used to access an active object.
IDLE task	A special low priority task which assumes control of the CPU when no other task is able to execute.
Instruction Pointer	A hardware register containing the address of the current instruction.
interface	A specification of the methodology used to connect multiple independent subsystems.
internal address	The address used to access dual-ported memory by the node which owns the memory.
interrupt	A hardware facility that causes the CPU to suspend execution, save its status, and transfer control to a specific location.
interrupt level	A mask used to by the CPU to determine which pending interrupts should be serviced. If a pending interrupt is below the current interrupt level, then the CPU does not recognize that interrupt.
Interrupt Service Routine	An ISR is invoked by the CPU to process a pending interrupt.
I/O	An acronym for Input/Output.
IP	An acronym for Instruction Pointer.
ISR	An acronym for Interrupt Service Routine.

kernel	In this document, this term is used as a synonym for executive.
list	A data structure which allows for dynamic addition and removal of entries. It is not statically limited to a particular size.
little endian	A data representation scheme in which the bytes composing a numeric value are arranged such that the least significant byte is at the lowest address.
local	An object which was created without the GLOBAL attribute set and is accessible only on the node it was created and resides upon. In a single processor configuration, all objects are local.
local operation	The manipulation of an object which resides on the same node as the calling task.
logical address	An address used by an application. In a system without memory management, logical addresses will equal physical addresses.
loosely-coupled	A multiprocessor configuration where shared memory is not used for communication.
major number	The index of a device driver in the Driver Address Table.
manager	A group of related RTEMS' directives which provide access and control over resources.
memory pool	Used interchangeably with heap.
message	A variable byte entity used to communicate between tasks. Messages are sent to message queues and stored in message buffers.
message buffer	A block of memory used to store messages.
message queue	An RTEMS object used to synchronize and communicate between tasks by transporting messages between sending and receiving tasks.
Message Queue Control Block	A data structure associated with each message queue used by RTEMS to manage that message queue.

minor number	A numeric value passed to a device driver, the exact usage of which is driver dependent.
mode	An entry in a task's control block that is used to determine if the task allows preemption, timeslicing, processing of signals, and the processor mode used by the task. If the mode specifies that the task is in supervisor mode, then an interrupt level is used.
MPCI	An acronym for Multiprocessor Communications Interface Layer.
multiprocessing	The simultaneous execution of two or more processes by a multiple processor computer system.
multiprocessor	A computer with multiple CPUs available for executing applications.
Multiprocessor Communications Interface Layer	A set of user-provided routines which enable the nodes in a multiprocessor system to communicate with one another.
Multiprocessor *Configuration* Table	A table which contains the information needed by RTEMS only when used in a multiprocessor configuration.
multitasking	The alternation of execution amongst a group of processes on a single CPU. A scheduling algorithm is used to determine which process executes at which time.
mutual exclusion	A term used to describe the act of preventing other tasks from accessing a resource simultaneously.
nested	A term used to describe an ASR that occurs during another ASR or an ISR that occurs during another ISR.
node	A term used to reference a processor running RTEMS in a multiprocessor system.
non-existent	The state occupied by an uncreated or deleted task.
numeric coprocessor	A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.

object — In this document, this term is used to refer collectively to tasks, message queues, partitions, regions, semaphores, and timers.

object-oriented — A term used to describe systems with common mechanisms for utilizing a variety of entities. Object-oriented systems shield the application from implementation details.

operating system — The software which controls all the computer's resources and provides the base upon which application programs can be written.

overhead — The portion of the CPUs processing power consumed by the operating system.

packet — A buffer which contains the messages passed between nodes in a multiprocessor system. A packet is the contents of an envelope.

partition — An RTEMS object which is used to allocate and deallocate fixed size blocks of memory from an dynamically specified area of memory.

Partition Control Block — A data structure associated with each partition used by RTEMS to manage that partition.

PC — An acronym for Program Counter.

pending — A term used to describe a task blocked waiting for an event, message, semaphore, or signal.

periodic task — A task which must execute at regular intervals and must comply with a hard deadline.

physical address — The actual hardware address of a resource.

poll — A mechanism used to determine if an event has occurred by periodically checking for a particular status. Typical events include arrival of data, completion of an action, and errors.

pool — A collection from which resources are allocated.

portability — A term used to describe the ease with which software can be rehosted on another computer.

posting	The act of sending an event, message, semaphore, or signal to a task.
preempt	The act of forcing a task to relinquish the processor and dispatching to another task.
priority	A mechanism used to represent the relative importance of a set of items.
processor utilization	The percentage of processor time used by a task or a set of tasks.
Program Counter	A hardware register containing the address of the current instruction.
Program Status Register	A microprocessor register typically containing the processor execution mode, interrupt level, trace mode, and condition codes.
proxy	An RTEMS control structure used to represent, on a remote node, a task which must block as part of a remote operation.
Proxy Control Block	A data structure associated with each proxy used by RTEMS to manage that proxy.
PSR	An acronym for Program Status Register.
PTCB	An acronym for Partition Control Block.
PXCB	An acronym for Proxy Control Block.
quantum	The application defined unit of time in which the processor is allocated.
queue	Alternate term for message queue. A data structure managed using the FIFO discipline.
QCB	An acronym for Message Queue Control Block.
ready	A task occupies this state when it is available to be given control of the CPU.
real-time	A term used to describe systems which are characterized by requiring deterministic response times to external stimuli.

	The external stimuli require that the response occur at a precise time or the response is incorrect.
reentrant	A term used to describe routines which do not modify themselves or global variables.
region	An RTEMS object which is used to allocate and deallocate variable size blocks of memory from a dynamically specified area of memory.
Region Control Block	A data structure associated with each region used by RTEMS to manage that region.
registers	Registers are locations physically located within a component, typically used for device control or general purpose storage.
remote	Any object that does not reside on the local node.
remote operation	The manipulation of an object which does not reside on the same node as the calling task.
return code	Also known as error code or return value.
resource	A hardware or software entity to which access must be controlled.
resume	Removing a task from the suspend state. If the task's state is ready following a call to the t_resume directive, then the task is available for scheduling.
return code	An value returned by RTEMS directives to indicate the completion status of the directive.
RNCB	An acronym for Region Control Block.
round-robin	A task scheduling discipline in which tasks of equal priority are executed in the order in which they are made ready.
RS-232	A standard for serial communications.
running	The state of a rate monotonic timer while it is being used to delineate a period. The timer exits this state by either expiring or being canceled.

schedule	The process of choosing which task should next enter the executing state.
schedulable	A set of tasks which can be guaranteed to meet their deadlines based upon a specific scheduling algorithm.
segments	Variable sized memory blocks allocated from a region.
semaphore	An RTEMS object which is used to synchronize tasks and provide mutually exclusive access to resources.
Semaphore Control Block	A data structure associated with each semaphore used by RTEMS to manage that semaphore.
shared memory	Memory which is accessible by multiple nodes in a multiprocessor system.
signal	An RTEMS provided mechanism to communicate asynchronously with a task. Upon reception of a signal, the ASR of the receiving task will be invoked.
signal set	A thirty-two bit entity which is used to represent a task's collection of pending signals and the signals sent to a task.
SMCB	An acronym for Semaphore Control Block.
soft real-time	A real-time system in which a missed deadline does not compromise the integrity of the system.
sporadic task	A task which executes at irregular intervals and must comply with a hard deadline. A minimum period of time between successive iterations of the task can be guaranteed.
SR	An acronym for Status Register.
stack	A data structure that is managed using a Last In First Out (LIFO) discipline. Each task has a stack associated with it which is used to store return information and local variables.
status code	Also known as error code or return value.
status register	A microprocessor register typically containing the processor execution mode, interrupt level, trace mode, and condition codes.

suspend	A term used to describe a task that is not competing for the CPU because it has had a t_suspend() directive.
synchronous	Related in order or timing to other occurrences in the system.
system call	In this document, this is used as an alternate term for directive.
target	The system on which the application will ultimately execute.
task	A logically complete thread of execution. The CPU is allocated among the ready tasks.
Task Control Block	A data structure associated with each task used by RTEMS to manage that task.
task switch	Alternate terminology for context switch. Taking control of the processor from one task and given to another.
TCB	An acronym for Task Control Block.
tick	The basic unit of time used by RTEMS. It is a user-configurable number of milliseconds. The current tick expires when the tm_tick directive is invoked.
tightly-coupled	A multiprocessor configuration system which communicates via shared memory.
timeout	An argument provided to a number of directives which determines the maximum length of time an application task is willing to wait for the directive to complete.
timer	An RTEMS object used to send events to the calling tasks at a later time.
Timer Control Block	A data structure associated with each timer used by RTEMS to manage that timer.
timeslicing	A task scheduling discipline in which tasks of equal priority are executed for a specific period of time before being preempted by another task.

timeslice	The application defined unit of time in which the processor is allocated.
TMCB	An acronym for Timer Control Block.
transient overload	A temporary rise in system activity which may cause deadlines to be missed. Rate Monotonic Scheduling can be used to determine if all deadlines will be met under transient overload.
user extensions	Software routines provided by the application to enhance the functionality of RTEMS.
User Extension Table	A table which contains the entry points for each user extensions.
User Initialization Task Table	A table which contains the information needed to create and start each of the user initialization tasks.
user-provided	Alternate term for user-supplied. This term is used to designate any software routines which must be written by the application designer.
user-supplied	Alternate term for user-provided. This term is used to designate any software routines which must be written by the application designer.
vector	Memory pointers used by the processor to fetch the address of routines which will handle various exceptions and interrupts.
wait queue	The list of tasks blocked pending the release of a particular resource. Message queues, regions, and semaphores have a wait queue associated with them.
yield	When a task voluntarily releases control of the processor.

RTEMS Directives

Initialization Manager

init_exec	Initialize RTEMS

Task Manager

t_create	Create a task
t_ident	Get ID of a task
t_start	Start a task
t_restart	Restart a task
t_delete	Delete a task
t_suspend	Suspend a task
t_resume	Resume a task
t_setpri	Set task priority
t_mode	Change current task's mode
t_getnote	Get task notepad entry
t_setnote	Set task notepad entry

Interrupt Manager

i_catch	Establish an ISR

Time Manager

tm_set	Set system date and time
tm_get	Get system date and time
tm_wkafter	Wake up after interval
tm_wkwhen	Wake up when specified
tm_evafter	Send event set after interval
tm_evwhen	Send event set when specified
tm_evevery	Send periodic event set
tm_delete	Delete timer event
tm_tick	Announce a clock tick

Semaphore Manager

sm_create	Create a semaphore
sm_ident	Get ID of a semaphore
sm_delete	Delete a semaphore
sm_p	Acquire a semaphore
sm_v	Release a semaphore

Message Manager

q_create	Create a queue
q_ident	Get ID of a queue
q_delete	Delete a queue
q_send	Put message at rear of a queue
q_urgent	Put message at front of a queue
q_broadcast	Broadcast N messages to a queue
q_receive	Receive message from a queue
q_flush	Flush all messages on a queue

Event Manager

ev_send	Send event to a task
ev_receive	Receive event condition

Signal Manager

as_catch	Establish an ASR
as_send	Send signal set to a task

Partition Manager

pt_create	Create a partition
pt_ident	Get ID of a partition
pt_delete	Delete a partition
pt_getbuf	Get buffer from a partition
pt_retbuf	Return buffer to a partition

Region Manager

rn_create	Create a region
rn_ident	Get ID of a region
rn_delete	Delete a region
rn_getseg	Get segment from a region
rn_retseg	Return segment to a region

Dual-Ported Memory Manager

dp_create	Create a port
dp_ident	Get ID of a port
dp_delete	Delete a port
dp_2internal	Convert external to internal address
dp_2external	Convert internal to external address

I/O Manager

de_init	Initialize a device driver
de_open	Open a device
de_close	Close a device
de_read	Read from a device
de_write	Write to a device
de_cntrl	Special device services

Fatal Error Manager

k_fatal	Invoke the fatal error handler

Rate Monotonic Manager

rm_create	Create a period
rm_cancel	Cancel a period
rm_delete	Delete a period
rm_period	Conclude current/Start next period

Multiprocessing Manager

mp_announce	Announce the arrival of a packet

INITIAL DISTRIBUTION LIST

	Copies
U.S. Army Materiel System Analysis Activity ATTN: AMXSY–MP (Herbert Cohen) Aberdeen Proving Ground, MD 21005	1
IIT Research Institute ATTN: GACIAC 10 W. 35th Street Chicago, IL 60616	1
Naval Weapons Center Missile Software Technology Office Code 3901C, ATTN: Mr. Carl W. Hall China Lake, CA 93555–6001	1
On–Line Applications Research 2227 Drake Avenue SW, Suite 10–F Huntsville, AL 35805	3
Louis H. Coglianese Advanced Technology IBM Corporation Federal Systems Company Route 17C, Mail Drop 0210 Owega, NY 13827	1
Kenneth Gregson MIT Lincoln Laboratory 244 Wood Street Lesington, MA 02173	1
John R. James Washington Engineering Division Intermetrics, Inc. 7918 Jones Branch Drive Suite 710 McLean, VA 22012	1
Keng Low SSC Lab M.S. 4002 2550 Beckleymeade Avenue Dallas, Texas 75237	1
Jeff Stewart Software Engineering Institute RM 5505 Carnegie Mellon University Pittsburgh, PA 15213	1

INITIAL DISTRIBUTION LIST (Continued)

	Copies
James M. Short ODDRE&E (R&AT) RM. 3D1089, The Pentagon Washington, D.C. 20301–3080	1
Connie Palmer McDonnel Douglas Computer & Software Technology Ctr MS: 3064285 St. Louis, MO 63133–0516	1
Chris Anderson Ada–9X Project Office P1/VTET Kirkland AFB, NM 87117–6008	1
Virginia Castor ODDRE&E RM. 3E118, The Pentagon Washington, D.C. 20301–3080	1
Sholom Cohen Software Engineering Institute Carnegie–Mellon University Pittsburgh, PA 15213	1
CEA Incorporated Blue Hills Office Park 150 Royall Street Suite 260, ATTN: Mr. John Shockro Canton, MA 01021	1
VITA 10229 N. Scottsdale Rd. Suite B, ATTN: Mr. Ray Alderman Scottsdale, AZ 85253	1
Westinghouse Electric Corp. P.O. Box 746–MS432 ATTN: Mr. Eli Soloman Baltimore, MD 21203	1
Dept. of Computer Science B–173 Florida State University ATTN: Dr. Ted Baker Tallahassee, FL 32306–4019	1

INITIAL DISTRIBUTION LIST (Continued)

		Copies
DSD Laboratories 75 Union Avenue ATTN: Mr. Roger Whitehead Studbury, MA 01776		1
AMSMI–RD		1
AMSMI–RD–GC,	Dr. Paul Jacobs	1
AMSMI–RD–GC–S,	Gerald E. Scheiman Phillip R. Acuff	1 4
AMSMI–RD–GC–N	Wanda M. Hughes	10
AMSMI–RD–BA		1
AMSMI–RD–BA–C3,	Bob Christian	1
AMSMI–RD–BA–AD,	Bruce Lewis	1
AMSMI–RD–SS		1
AMSMI–RD–CS–R		15
AMSMI–RD–CS–T		1
AMSMI–RD–GC–IP,	Mr. Fred H. Bush	1
CSSD–CR–S,	Mr. Frank Poslajko	1
SFAE–FS–ML–TM,	Mr. Frank Gregory	1
SFAE–AD–ATA–SE,	Mr. Julian Cothran Mr. John Carter	1 1

U.S. GOVERNMENT PRINTING OFFICE 1993 533-143 /80119